

# FUNDAMENTOS DE

# C# 3.0

Cubre:

Tipos de datos, operadores y declaraciones  
de control • Clases, objetos y métodos •

Propiedades, indexadores y eventos •  
LINQ, expresiones lambda, genéricos •

Y mucho más

Herbert Schildt



# Fundamentos de C# 3.0

## Acerca del autor

**Herb Schildt** es una autoridad en C#, Java, C y C++. Sus libros de programación han vendido más de 3.5 millones de copias en todo el mundo y han sido traducidos a los idiomas más importantes del planeta. Sus aclamados títulos incluyen: *Java Manual de referencia séptima edición*, *Fundamentos de Java*, *Java: Soluciones de programación* y *C++: Soluciones de programación*. Aunque se interesa en todas las facetas de la computación, su principal foco de atención son los lenguajes de cómputo, incluyendo compiladores, intérpretes y lenguajes para el control de robots. También muestra un especial interés en la estandarización de los lenguajes de cómputo. Schildt imparte cátedra a nivel superior y postgrado en la Universidad de Illinois. Puede ser localizado en su oficina de consultas en el teléfono (217) 586-4683 y su página Web es [www.HerbSchildt.com](http://www.HerbSchildt.com).

## Acerca del editor técnico

**Eric Lippert** es desarrollador senior del equipo de desarrollo de C# en Microsoft.

# Fundamentos de C# 3.0

**Herbert Schildt**

**Traducción**

**Luis Antonio Magaña Pineda**

*Traductor profesional*



MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • MADRID • NUEVA YORK  
SAN JUAN • SANTIAGO • SÃO PAULO • AUCKLAND • LONDRES • MILÁN • MONTREAL  
NUEVA DELHI • SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

**Director editorial:** Fernando Castellanos Rodríguez  
**Editor:** Miguel Ángel Luna Ponce  
**Supervisor de producción:** Zeferino García García

## FUNDAMENTOS DE C# 3.0

Prohibida la reproducción total o parcial de esta obra,  
por cualquier medio, sin la autorización escrita del editor.



DERECHOS RESERVADOS © 2010, respecto a la primera edición en español por  
McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V.

*A Subsidiary of The McGraw-Hill Companies, Inc.*

Corporativo Punta Santa Fe,  
Prolongación Paseo de la Reforma 1015, Torre A  
Piso 17, Colonia Desarrollo Santa Fe,  
Delegación Álvaro Obregón  
C.P. 01376, México, D.F.  
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

**ISBN: 978-607-15-0236-0**

Translated from the 2nd English edition of

*C# 3.0: A Beginner's Guide*

By: Herbert Schildt

Copyright © 2009 by The McGraw-Hill Companies. All rights reserved.

ISBN: 978-0-07-158830-0

1234567890

109876543210

Impreso en México

*Printed in Mexico*

# Contenido

PREFACIO .....	xiii
<b>1 Fundamentos de C# .....</b>	<b>1</b>
El árbol genealógico de C# .....	2
C: el principio de la era moderna de la programación .....	3
La creación de POO y C++.....	3
El surgimiento de Internet y Java .....	4
La creación de C#.....	5
La evolución de C#.....	6
Cómo se relaciona C# con .NET Framework.....	6
¿Qué es .NET Framework?.....	7
Cómo funciona el lenguaje común de tiempo de ejecución.....	7
Código controlado <i>vs.</i> código sin control.....	8
La especificación del lenguaje común .....	9
Programación orientada a objetos.....	9
Encapsulado .....	10
Polimorfismo .....	10
Herencia .....	11
Crear, compilar y ejecutar tu primer programa en C# .....	12
Obtener un compilador C# 3.0 .....	12
Utilizar el IDE de Visual Studio .....	13
Usar csc.exe, el compilador de línea de comandos de C# .....	17
El primer programa ejemplo línea por línea.....	18
Manejo de errores de sintaxis.....	21
Una pequeña variación.....	22
Usar una variable .....	22
El tipo de dato Doble .....	25
Prueba esto: Convertir Fahrenheit a Celsius .....	27
Dos declaraciones de control .....	28
La declaración if .....	28
El loop for.....	30
Utilizar bloques de código .....	31
Punto y coma, y posiciones .....	32
Prácticas de indentado .....	33
Prueba esto: mejorar el programa de conversión de temperatura.....	34
Las palabras clave de C# .....	35
Identificadores .....	36
La biblioteca de clases de C# .....	37
<b>2 Introducción a los tipos de datos y operadores .....</b>	<b>39</b>
Por qué son importantes los tipos de datos .....	40
Tipos de valor C# .....	40
Enteros .....	41

Tipos de punto flotante . . . . .	44
El tipo decimal . . . . .	44
Caracteres . . . . .	46
El tipo bool . . . . .	47
Algunas opciones para datos de salida . . . . .	48
Prueba esto: Hablar a Marte . . . . .	49
LITERALES . . . . .	51
LITERALES hexadecimales . . . . .	52
Secuencias de caracteres de escape . . . . .	52
LITERALES de cadena de caracteres . . . . .	53
UN ANÁLISIS profundo de las variables . . . . .	55
Inicializar una variable . . . . .	56
Inicialización dinámica . . . . .	56
Variables de tipo implícito . . . . .	57
EL alcance y tiempo de vida de las variables . . . . .	58
OPERADORES . . . . .	61
OPERADORES aritméticos . . . . .	61
INCREMENTO y decremento . . . . .	62
OPERADORES lógicos y de relación . . . . .	64
OPERADORES lógicos de circuito corto . . . . .	66
Prueba esto: Mostrar una tabla de verdad para los operadores lógicos . . . . .	67
EL operador de asignación . . . . .	69
Combinar asignaciones . . . . .	69
CONVERSIÓN de tipo en las asignaciones . . . . .	70
Transformar tipos incompatibles . . . . .	72
Prioridad de los operadores . . . . .	73
CONVERSIÓN de tipo dentro de expresiones . . . . .	73
Espacios en blanco y paréntesis . . . . .	77
Prueba esto: Calcular los pagos regulares de un préstamo . . . . .	77
<b>3 Declaraciones para el control del programa . . . . .</b>	<b>81</b>
Ingresar caracteres desde el teclado . . . . .	82
La declaración if . . . . .	83
if anidados . . . . .	85
La escalera if-else-if . . . . .	86
La declaración switch . . . . .	87
Declaraciones switch anidadas . . . . .	91
Prueba esto: Comenzar a construir un sistema de ayuda C# . . . . .	92
El loop for . . . . .	94
Algunas variantes del loop for . . . . .	96
Declarar variables de control de reiteración dentro del loop for . . . . .	99
El loop while . . . . .	100
El loop do-while . . . . .	102
Prueba esto: Mejorar el sistema de ayuda C# . . . . .	104
Usar break para salir de un loop . . . . .	106
Utilizar continue . . . . .	109
La instrucción goto . . . . .	109
Prueba esto: Finalizar el programa de ayuda C# . . . . .	111
Loops anidados . . . . .	115

---

<b>4 Introducción a las clases, objetos y métodos .....</b>	<b>119</b>
Fundamentos de las clases .....	120
La forma general de una clase .....	121
Definir una clase .....	122
Cómo se crean los objetos .....	126
Variables de referencia y asignaciones .....	126
Métodos .....	127
Añadir un método a la clase vehículo .....	128
Regresar de un método .....	130
Regresar un valor como respuesta .....	131
Usar parámetros .....	134
Añadir un método con parámetros a vehículo .....	135
Prueta esto: Crear una clase Ayuda .....	137
Constructores .....	143
Constructores parametrizados .....	144
Añadir un constructor a la clase Vehículo .....	145
El operador new revisado .....	146
Recolección de basura y destructores .....	147
Destructores .....	148
La palabra clave this .....	148
<b>5 Más tipos de datos y operadores.....</b>	<b>153</b>
Arreglos .....	154
Arreglos unidimensionales .....	155
Prueta esto: Organizar un arreglo .....	159
Arreglos multidimensionales .....	161
Arreglos bidimensionales .....	161
Arreglos de tres o más dimensiones .....	162
Inicializar arreglos multidimensionales .....	162
Arreglos descuadrados .....	163
Asignar referencias a arreglos .....	165
Utilizar la propiedad de longitud con arreglos .....	167
Crear un tipo de arreglo implícito .....	169
Prueta esto: Crear una clase sencilla de estructura en cola .....	170
El loop foreach .....	174
Cadenas de caracteres .....	177
Construir un objeto string .....	177
Operadores de objetos string .....	178
Arreglos de objetos string .....	180
Los objetos string son inmutables .....	181
Los operadores bitwise .....	182
Los operadores bitwise AND, OR, XOR y NOT .....	183
Los operadores de traslado .....	187
Asignaciones para mezclar bitwise .....	189
Prueta esto: Crear una clase que muestre los bits .....	189
El operador ? .....	192
<b>6 Un análisis profundo de los métodos y las clases.....</b>	<b>197</b>
Controlar el acceso a los miembros de la clase .....	198
Especificadores de acceso de C# .....	199
Prueta esto: Mejorar la clase sencilla de estructura en cola .....	203

Transmitir una referencia de objeto a un método.....	205
Cómo se transmiten los argumentos.....	206
Utilizar los parámetros ref y out.....	208
Usar ref.....	209
Usar out.....	211
Utilizar un número indeterminado de argumentos.....	213
Objetos como respuesta .....	216
Sobrecargar un método.....	218
Sobrecargar constructores.....	224
Invocar un constructor sobrecargado a través de this.....	226
Prueba esto: Sobrecargar un constructor ColaSimple .....	228
El método Main() .....	231
Regresar valores de Main() .....	231
Transmitir argumentos a Main() .....	231
Recursión .....	234
Comprender static .....	236
Constructores y clases static.....	238
Prueba esto: Acomodo rápido.....	239
<b>7 Sobrecarga de operadores, indexadores y propiedades .....</b>	<b>245</b>
Sobrecarga de operadores.....	246
Los formatos generales de un método operador.....	247
Sobrecargar operadores binarios.....	247
Sobrecargar operadores unitarios.....	250
Añadir flexibilidad .....	254
Sobrecargar los operadores de relación .....	259
Consejos y restricciones para la sobrecarga de operadores .....	261
Indexadores .....	262
Indexadores multidimensionales .....	267
Restricciones para los indexadores.....	269
Propiedades .....	270
Propiedades autoimplementadas.....	273
Restricciones de las propiedades .....	274
Utilizar un modificador de acceso con un accesador.....	274
Prueba esto: Crear una clase conjunto .....	277
<b>8 Herencia .....</b>	<b>287</b>
Bases de la herencia .....	288
Acceso a miembros y herencia.....	291
Utilizar acceso protegido .....	294
Constructores y herencia .....	296
Invocar constructores de la clase base .....	298
Herencia y ocultamiento de nombres .....	302
Utilizar base para accesar un nombre oculto .....	303
Prueba esto: Extender la clase Vehículo .....	305
Crear una jerarquía multinivel .....	308
¿Cuándo se invocan los constructores? .....	311
Referencias a la clase base y objetos derivados .....	313
Métodos virtuales y anulación .....	315
¿Por qué métodos anulados? .....	318
Aplicar métodos virtuales.....	318

---

Utilizar clases abstractas .....	322
Utilizar sealed para prevenir la herencia .....	326
La clase object .....	327
Encajonar y desencajonar .....	329
<b>9 Interfaces, estructuras y enumeraciones .....</b>	<b>333</b>
Interfaces .....	334
Implementar interfaces .....	335
Utilizar referencias de interfaz .....	339
Pruéba esto: Crear una interfaz de orden en cola .....	341
Interfaz para propiedades .....	347
Interfaz para indexadores .....	349
Las interfaces pueden heredarse .....	351
Implementaciones explícitas .....	353
Estructuras .....	355
Enumeraciones .....	357
Inicializar una enumeración .....	359
Especificar el tipo subyacente de una enumeración .....	360
<b>10 Manejo de excepciones.....</b>	<b>361</b>
La clase System.Exception .....	362
Fundamentos del manejo de excepciones .....	363
Utilizar try y catch .....	363
Un sencillo ejemplo de excepción .....	364
Un segundo ejemplo de excepción .....	365
Las consecuencias de una excepción sin atrapar .....	366
Las excepciones te permiten controlar errores con elegancia .....	368
Utilizar múltiples cláusulas catch.....	369
Atrapar todas las excepciones .....	370
Los bloques try se pueden anidar .....	370
Lanzar una excepción .....	372
Volver a lanzar una excepción .....	373
Utilizar finally .....	374
Análisis detallado de las excepciones .....	376
Excepciones de uso común .....	378
Derivar clases de excepción .....	378
Atrapar excepciones de clases derivadas .....	380
Pruéba esto: Añadir excepciones a la clase orden en cola .....	382
Utilizar checked y unchecked .....	386
<b>11 Utilizar E/S .....</b>	<b>391</b>
E/S de C# está construido sobre flujo de datos .....	392
Flujo de bytes y flujo de caracteres .....	392
Flujos predefinidos .....	393
Las clases de flujo .....	393
La clase Stream .....	393
Las clases de flujo de bytes .....	394
Las clases envueltas de flujo de caracteres .....	394
Flujos binarios .....	396

E/S de la consola .....	397
Leer datos de entrada de la consola .....	397
Escribir datos de salida de la consola .....	398
FileStream y E/S de archivos orientados a byte .....	400
Abrir y cerrar un archivo .....	400
Leer bytes de un FileStream .....	402
Escribir en un archivo .....	404
E/S de un archivo basado en caracteres .....	406
Utilizar StreamWriter .....	406
Utilizar StreamReader .....	409
Redireccionar los flujos estándar .....	410
Prueba esto: Crear una utilidad para comparar archivos .....	412
Leer y escribir datos binarios .....	414
Escritor binario .....	414
Lector binario .....	415
Mostrar E/S binaria .....	416
Archivos de acceso aleatorio .....	418
Convertir cadenas numéricas en su representación interna .....	420
Prueba esto: Crear un sistema de ayuda basado en disco .....	425
<b>12 Delegados, eventos y nomenclaturas.....</b>	<b>431</b>
Delegados .....	433
Utilizar métodos de instancia como delegados .....	436
Distribución múltiple .....	437
Por qué delegados .....	439
Métodos anónimos .....	440
Eventos .....	443
Un ejemplo de evento de distribución múltiple .....	445
Utilizar métodos anónimos con eventos .....	448
Nomenclaturas .....	450
Declarar una nomenclatura .....	450
using .....	452
Una segunda forma de using .....	454
Las nomenclaturas son aditivas .....	455
Las nomenclaturas pueden anidarse .....	456
La nomenclatura global .....	458
Prueba esto: Colocar Set en una nomenclatura .....	459
<b>13 Genéricos.....</b>	<b>463</b>
¿Qué son los genéricos? .....	465
Fundamentos de los genéricos .....	465
Los tipos genéricos difieren con base en sus argumentos de tipo .....	468
Los genéricos mejoran la seguridad de los tipos .....	468
Una clase genérica con dos parámetros de tipo .....	471
Tipos limitados .....	473
Utilizar una limitación clase base .....	474
Utilizar una limitación para establecer una relación entre dos parámetros de tipo .....	476
Utilizar una interfaz limitada .....	477
Utilizar el constructor new ( ) limitado .....	479
Las limitaciones del tipo referencia y del tipo valor .....	481
Utilizar limitaciones múltiples .....	484

---

Crear un valor por omisión de un parámetro de tipo .....	485
Estructuras genéricas .....	487
Métodos genéricos .....	488
Utilizar argumentos de tipo explícito para invocar un método genérico .....	491
Utilizar una limitación con un método genérico.....	491
Delegados genéricos.....	492
Interfaces genéricas .....	494
Prueba esto: Crear un orden de cola genérico .....	498
<b>14 Introducción a LINQ .....</b>	<b>505</b>
¿Qué es LINQ? .....	507
Fundamentos de LINQ.....	507
Un query sencillo .....	508
Un query puede ser ejecutado más de una vez.....	510
Cómo se relacionan los tipos de datos en un query .....	511
El formato general de un query .....	512
Filtrar valores con where .....	513
Ordenar resultados con orderby .....	514
Una mirada cercana a select.....	516
Agrupar resultados con group .....	519
Utilizar into para crear una continuidad.....	521
Utilizar let para crear una variable en un query .....	523
Unir dos secuencias con join .....	525
Tipos anónimos e inicializadores de objetos .....	528
Crear una conjunción de grupo .....	531
Los métodos de query y las expresiones lambda.....	534
Los métodos de query básicos .....	534
Expresiones lambda .....	535
Crear consultas utilizando los métodos de query .....	536
Más extensiones de métodos relacionados con query .....	539
Aplazamiento <i>vs.</i> inmediatez en la ejecución del query .....	541
Una mirada más cercana a la extensión de métodos .....	542
Una mirada más cercana a las expresiones lambda .....	544
Expresión lambda .....	545
Declaraciones lambda.....	546
Prueba esto: Utilizar expresiones lambda para implementar controladores de eventos .....	547
<b>15 El preprocesador, RTTI, tipos anulables y otros temas avanzados .....</b>	<b>553</b>
El preprocesador .....	554
#define .....	555
#if y #endif .....	555
#else y #elif.....	557
#undef.....	559
#error.....	559
#warning .....	559
#line .....	560
#region y #endregion .....	560
#pragma .....	560

Identificación del tipo tiempo de ejecución .....	561
Probar un tipo con is.....	561
Utilizar as .....	562
Utilizar typeof.....	562
Tipos anulables.....	563
El operador ??.....	565
Objetos anulables y los operadores relacionales y lógicos.....	566
Código inseguro .....	567
Una breve mirada a los punteros.....	567
La palabra clave unsafe.....	570
Utilizar fixed.....	570
Atributos.....	572
El atributo Condicional.....	572
El atributo obsolete .....	573
Operadores de conversión .....	574
Una breve introducción a las colecciones .....	578
Bases de las colecciones.....	578
Un caso de estudio de colecciones: crear un arreglo dinámico .....	580
Prueba esto: Utilizar la colección Queue<T>.....	583
Otras palabras clave .....	586
El modificador de acceso interno.....	586
sizeof.....	586
lock .....	586
readonly .....	587
stackalloc .....	587
La declaración using.....	588
const y volatile .....	589
El modificador partial.....	589
yield .....	591
extern .....	592
¿Qué sigue? .....	592
<b>A Respuestas a los autoexámenes .....</b>	<b>595</b>
Capítulo 1: Fundamentos de C#.....	596
Capítulo 2: Introducción a los tipos de datos y operadores .....	597
Capítulo 3: Declaraciones para el control del programa .....	598
Capítulo 4: Introducción a las clases, objetos y métodos.....	600
Capítulo 5: Más tipos de datos y operadores .....	601
Capítulo 6: Un análisis profundo de los métodos y las clases .....	603
Capítulo 7: Sobrecarga de operadores, indexadores y propiedades.....	607
Capítulo 8: Herencia.....	609
Capítulo 9: Interfaces, estructuras y enumeraciones .....	610
Capítulo 10: Manejo de excepciones .....	612
Capítulo 11: Utilizar E/S .....	614
Capítulo 12: Delegados, eventos y nomenclaturas.....	617
Capítulo 13: Genéricos.....	618
Capítulo 14: Introducción a LINQ .....	618
Capítulo 15: El preprocesador, RTTI, tipos anulables y otros temas avanzados .....	619
<b>Índice .....</b>	<b>621</b>

# Prefacio

**E**n una época donde “la computadora es la red”, la tecnología .NET Framework se ha convertido en un ambiente de desarrollo líder para la creación de código. El principal lenguaje para el desarrollo en .NET es C#. Por ello, si la programación .NET está en tu futuro, has elegido aprender el lenguaje indicado.

Más allá de su uso para programar en .NET, C# es importante por otras razones. Sus características innovadoras están rediseñando el mundo de la programación, cambiando la manera de escribir el código y permitiendo que las soluciones sean enmarcadas en nuevas condiciones. Por ello, C# está ayudando a definir el futuro rumbo de la programación. Como resultado, la destreza en C# ha dejado de ser una opción para el programador profesional, ahora es una necesidad.

El propósito de este libro es enseñarte los fundamentos de la programación con C#. Utiliza una metodología paso a paso, complementada con numerosos ejemplos y autoexámenes. No da por hecho que tengas experiencia previa en la programación. Este libro comienza con lo básico, por ejemplo, cómo compilar y ejecutar un programa en C#. Después aborda las palabras clave, características y constructores de los que consta este lenguaje. Cuando finalices tendrás un firme dominio de los fundamentos de la programación en C#.

Como lo saben todos los programadores, nada permanece estático durante mucho tiempo en el mundo de la programación. C# no es la excepción. Desde su creación en 2000 el lenguaje ha tenido dos grandes revisiones, cada una de las cuales le ha añadido nuevas y significativas características. En el momento en que se escribe este libro, la versión actual de C# es la 3.0 y ésa es la que se describe en el texto. Por ello, este libro abarca las características más recientes de C#, incluyendo el Lenguaje Query Integrado (LINQ) y expresiones lambda.

Por supuesto, esta guía para principiantes es sólo el punto de arranque. C# es un lenguaje muy amplio e incluye mucho más que las palabras clave y la sintaxis que lo define. También incluye el uso de un conjunto muy sofisticado de bibliotecas llamado Biblioteca de Clases .NET Framework.

Esta biblioteca es muy extensa y un análisis completo requería un libro aparte. Aunque muchas de las clases definidas en esta biblioteca se abordan en el presente libro, la mayoría de ellas han quedado fuera por cuestiones de espacio. Sin embargo, para ser un programador C# de alto rango es necesario dominar esta biblioteca. Después de terminar este libro tendrás los conocimientos necesarios para explorar la Biblioteca de Clases y todos los demás aspectos de C#.

## Cómo está organizado este libro

Este libro presenta un tutorial estructurado en el cual cada sección se construye con lo aprendido en la anterior. Son 15 capítulos, cada uno aborda un aspecto de C#. Este libro es único porque incluye varios elementos especiales que ayudan a organizar y reforzar los conocimientos que vas adquiriendo.

### Habilidades y conceptos clave

Cada capítulo inicia con una lista de las habilidades y los conceptos clave que aprenderás en esa parte.

### Autoexamen

Cada capítulo concluye con un autoexamen que te permite medir los conocimientos adquiridos. Las respuestas se encuentran en el Apéndice.

### Pregunta al experto

Encontrarás cuadros didácticos “Pregunta al experto” dispersos a lo largo del texto. Contienen información adicional o comentarios de interés sobre el tema abordado. Están redactados en formato pregunta – respuesta.

### Prueba esto

Cada capítulo contiene una o más secciones “Prueba esto”. En ellas se presentan ejemplos paso a paso que te enseñan a aplicar los conocimientos adquiridos.

## No se requiere experiencia previa en programación

Este libro no requiere experiencia previa en programación. Así, aunque nunca antes hayas programado, puedes utilizar este libro. Por supuesto, en esta época es probable que la mayoría de los lectores tengan al menos un poco de experiencia en la programación. Para muchos, la experiencia previa puede estar relacionada con C++ o Java. Como verás, C# está relacionado con ambos lenguajes. Por lo mismo, si ya conoces C++ o Java podrás aprender C# con mayor facilidad.

## Software requerido

Para compilar y ejecutar los programas de este libro necesitarás la versión 2008 (o posterior) de Visual Studio que soporte C#. Una buena opción es la Visual Studio C# 2008 Express Edition porque puede ser adquirida gratuitamente de la estación Web de Microsoft. Todo el código que aparece en este libro fue probado utilizando ese compilador gratuito. Por supuesto, el ambiente .Net Framework debe estar instalado en tu computadora.

## No lo olvides: código en Web

El código fuente de todos los ejemplos y proyectos que aparecen en este libro está disponible de manera gratuita a través de Web en <http://www.mcgraw-hill-educacion.com>. Realiza una búsqueda por autor, título de la obra o ISBN y bajo la portada de la obra encontrarás el link de descarga.

# Más de Herbert Schildt

*Fundamentos de C# 3.0* es sólo uno de los muchos libros sobre programación escritos por Herbert Schildt. He aquí algunos otros que encontrarás de tu interés.

Para continuar tu aprendizaje sobre C#, te sugerimos:

*C# 3.0: The Complete Reference*

Para aprender sobre Java, te recomendamos:

*Java Manual de referencia séptima edición*

*Fundamentos de Java*

*Java soluciones de programación*

*The Art of Java*

*Swing: A Beginner's Guide*

Para aprender C++, encontrarás particularmente útiles los siguientes libros:

*C++ Soluciones de programación*

*C++: The Complete Reference*

*C++: A Beginner's Guide*

*C++ from the Ground Up*

*STL Programming from the Ground Up*

*The Art of C++*

Si quieres aprender el lenguaje C, el siguiente título será de interés:

*C: The Complete Reference*

**Cuando necesites respuestas sólidas y rápidas, consulta a Herbert Schildt,  
una autoridad reconocida en programación.**

# Capítulo 1

## Fundamentos de C#

### Habilidades y conceptos clave

- La historia detrás de C#
  - Cómo se relaciona C# con .NET Framework y cómo lo utiliza
  - Los tres principios de la programación orientada a objetos
  - Crear, compilar y ejecutar programas C#
  - Variables
  - Las declaraciones **if** y **for**
  - Bloques de código
  - Las palabras clave de C#
- 

**L**a búsqueda de un lenguaje de programación perfecto es tan antigua como la programación misma. En esta búsqueda, C# es el actual estandarte. Creado por Microsoft para soportar su ambiente de desarrollo .NET Framework, C# batió las características de tiempo estimado con agudas innovaciones y proporcionó un medio altamente útil y eficiente para escribir programas para este ambiente de cómputo moderno empresarial. A lo largo de este libro aprenderás a programar utilizandoarlo.

El propósito de este capítulo es presentar una introducción a C#, incluyendo las fuerzas que llevaron a su creación, la filosofía de su diseño y varias de sus más importantes características. La parte más ardua de aprender un lenguaje de programación es, por mucho, el hecho de que ningún elemento existe de manera aislada. En lugar de ello, los componentes del lenguaje funcionan en conjunto. Es esta interacción lo que dificulta abordar un aspecto de C# sin involucrar otros. Para solucionar este problema, este capítulo proporciona una breve panorámica de varias características de C#, incluyendo la forma general de los programas desarrollados con este lenguaje, dos declaraciones de control y varios operadores. No se enfoca en los detalles, sino en los conceptos generales de uso común de cualquier programa C#.

En este momento la versión más reciente de C# es la 3.0, y es ésa la versión que se enseña en este libro. Por supuesto, mucha de la información aquí presentada se aplica a todas las versiones de C#.

### El árbol genealógico de C#

Los lenguajes de programación no existen en el vacío. Lejos de eso, se relacionan unos con otros; cada nuevo lenguaje está influenciado de una manera u otra por los anteriores. En un proceso consanguíneo, las características de un lenguaje son adoptadas por otro, alguna innovación es integrada a un contexto ya existente o un constructor obsoleto se elimina. De esta manera, los lenguajes evolucionan y el arte de la programación avanza. C# no es la excepción.

C# hereda un rico legado de programación. Es descendiente directo de dos de los más exitosos lenguajes de programación: C y C++. También está emparentado cercanamente con otro: Java. Comprender la naturaleza de estas relaciones es indispensable para entender la esencia de C#. Así, comenzamos nuestro análisis de C# colocándolo en el contexto histórico de estos tres lenguajes.

## C: el principio de la era moderna de la programación

La creación de C marca el inicio de la era moderna de la programación. C fue inventado por Dennis Ritchie en la década de 1970 en una computadora DEC PDP-11 que utilizaba el sistema operativo UNIX. Aunque algunos lenguajes anteriores, en particular Pascal, habían conseguido éxitos significativos, fue C el que estableció el paradigma que aún hoy marca el rumbo de la programación.

C creció a partir de la revolución de la *programación estructurada* de la década de 1960. Los lenguajes estructurados se definen por su rico conjunto de declaraciones de control bien diseñadas, subrutinas con variables locales, bloques de código y otras mejoras que facilitan la organización y el mantenimiento del programa. Aunque otros lenguajes de la época tenían características similares, C las implementó utilizando una sintaxis clara y fácil de utilizar. También aplicó una filosofía que ponía al programador, en lugar del lenguaje, como centro del proceso de desarrollo. Como resultado, C ganó rápidamente muchos seguidores. Se convirtió en el lenguaje de programación estructurada dominante a finales de la década de 1970, en la de 1980 y hoy en día sigue vigente.

## La creación de POO y C++

Para finales de la década de 1970 el tamaño de muchos proyectos estaba en los límites de lo que las metodologías de la programación estructurada y el lenguaje C podían manejar, a pesar de su utilidad. Los programas simplemente se volvían demasiado grandes. Para manejar este problema comenzó a emerge una nueva manera de programar. Este método es llamado *programación orientada a objetos* (POO para abreviar). Los programadores podían manejar programas más grandes utilizando POO. El problema era que C, el lenguaje más popular de la época, no ofrecía soporte para el nuevo método. El deseo de una versión de C orientada a objetos llevó, en el último de los casos, a la creación de C++.

C++ fue creado por Bjarne Stroustrup, cuyos inicios datan de 1979 en los Laboratorios Bell de Murray Hill, Nueva Jersey. Inicialmente llamó al nuevo lenguaje “C con Clases”, pero en 1983 el nombre fue reemplazado por el actual. C++ contiene todo el lenguaje C, por lo que el primero está fundamentado y construido enteramente sobre el segundo. La mayoría de las características que Stroustrup añadió a C estuvieron diseñadas para que soportara la programación orientada a objetos. En esencia, C++ es la versión POO de C.

Al construir sobre los cimientos de C, Stroustrup proporcionó un camino llano para la transición hacia POO. En vez de tener que aprender por completo un lenguaje nuevo, un programador de C sólo necesitaba aprender unas cuantas características nuevas para alcanzar los beneficios de la metodología orientada a objetos. Esto facilitó el cambio de la programación estructurada a la POO a legiones de programadores. Como resultado, para la segunda mitad de la década de 1990 C++ se convirtió en el lenguaje dominante para el desarrollo de código de alto rendimiento.

Es indispensable comprender que la invención de C++ no fue un intento por crear un nuevo lenguaje de programación. En lugar de eso, fue una mejora a un lenguaje que ya era exitoso. Este acercamiento al desarrollo de lenguajes (comenzar con un lenguaje existente y avanzar) estableció el curso que se sigue hoy en día.

## El surgimiento de Internet y Java

El siguiente avance sustancial en los lenguajes de programación fue Java. La creación de Java, que originalmente llevó el nombre Oak (roble), comenzó en 1991 en los laboratorios de Sun Microsystems. La principal fuerza impulsora detrás del diseño de Java fueron James Gosling, Patrick Naughton, Chris Warth y Ed Frank; Mike Sheridan también participó.

Java es un lenguaje estructurado y orientado a objetos, con sintaxis y filosofía derivada de C++. Los aspectos innovadores de Java estuvieron dirigidos no tanto hacia avances en el arte de la programación (aunque tuvo logros en esa materia), sino hacia modificaciones en el ambiente de programación. Antes de la oleada de Internet, la mayoría de los programas eran escritos, compilados y enfocados hacia una CPU específica y un sistema operativo en particular. Es verdad que los programadores siempre quisieron reutilizar su código, pero la posibilidad de migrar con facilidad un programa de un ambiente a otro implicaba muchos problemas. Sin embargo, con el surgimiento de Internet, donde diferentes tipos de CPU y sistemas operativos se conectan, el viejo problema de la portabilidad se convirtió en un aspecto de enorme importancia. Se requería un nuevo lenguaje para solucionar el problema, y ese nuevo lenguaje fue Java.

Java consigue la portabilidad al traducir el código fuente del programa a un lenguaje intermedio llamado *bytecode*. Este bytecode era ejecutado luego por la Máquina Virtual de Java (JVM, por sus siglas en inglés). Por tanto, un programa escrito en Java podía ejecutarse en cualquier ambiente para el cual estuviera disponible JVM. De la misma manera, dado que JVM es fácil de implementar, desde el inicio estaba disponible para una gran cantidad de ambientes.

Además de la necesidad de portabilidad, había un segundo problema fundamental que debía resolverse antes de que la programación basada en Internet se convirtiera en una realidad. Este problema era la seguridad. Como lo saben todos los usuarios de Internet, los virus representan una amenaza potencial, seria y constante. ¿De qué servirían los programas portables si no se podía confiar en ellos? ¿Quién querría correr el riesgo de ejecutar un programa descargado por Internet? Bien podría contener código malicioso. Por fortuna, la solución al problema de la seguridad también se encontró en JVM y el bytecode. Dado que JVM ejecuta el bytecode, tiene control total sobre el programa y puede evitar que el programa escrito en Java haga algo que no se supone que debería hacer. De esta manera, JVM y el bytecode solucionaron ambos problemas, tanto la portabilidad como la seguridad.

Es indispensable comprender que el uso de bytecode de Java difiere radicalmente de C y de C++, los cuales eran compilados casi siempre en código de máquina ejecutable. El código de máquina está amarrado a una CPU y a un sistema operativo específicos. De tal forma que si quisieras ejecutar un programa escrito en C o C++ en una computadora diferente, sería necesario que lo volvieras a compilar para el código de máquina específico del nuevo ambiente. Para crear un programa en C o C++ que se ejecutara en varios ambientes, sería necesario compilar diferentes versiones ejecutables del mismo. Esto no sólo era impráctico, también resultaba muy caro. El uso de un lenguaje intermedio por parte de Java fue una solución elegante y económica. También fue una solución que C# adaptaría para sus propios fines.

Como se mencionó, Java es descendiente de C y de C++. Aunque el código de Java no es compatible de manera ascendente ni descendente con C ni con C++, su sintaxis es lo bastante similar para que una gran cantidad de programadores de ambos lenguajes se trasladaran a Java con un esfuerzo mínimo. Más aún, dado que Java se construyó y mejoró un paradigma existente,

Gosling y los demás tuvieron la libertad de enfocar su atención en las características innovadoras del lenguaje. Así como Stroustrup no tuvo que “reinventar la rueda” cuando creó C++, Gosling no necesitó crear un lenguaje completamente nuevo cuando desarrolló Java. Más aún, con la creación de Java, C y C++ se convirtieron en los cimientos aceptados sobre los cuales se podían construir nuevos lenguajes de programación.

## La creación de C#

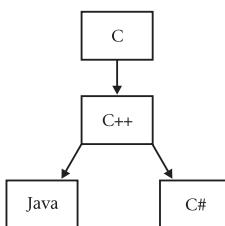
Java ofrece una solución exitosa a los temas concernientes a la portabilidad y la seguridad en el ambiente Internet, pero carece de algunas características importantes. Una de ellas es la *interoperabilidad de lenguajes cruzados*, también llamada *programación de lenguajes mixtos*. Se refiere a la posibilidad de que el código producido por un lenguaje pueda trabajar en conjunto y fácilmente con el código producido por otro lenguaje. La interoperabilidad de lenguajes cruzados es crucial para la creación de software de grandes dimensiones y amplia distribución. También es deseable para la programación de componentes de software, porque el componente más valioso es aquel que se puede utilizar en la más amplia variedad de lenguajes de cómputo y en el mayor número posible de ambientes operativos.

Otra característica de la que carece Java es su integración total con la plataforma Windows. Aunque los programas creados en Java pueden ejecutarse en el ambiente Windows (siempre y cuando se encuentre instalada la Máquina Virtual Java), ni uno ni otro se acoplan completamente. Dado que Windows es el sistema operativo más utilizado en el mundo, la falta de soporte directo para Windows es una carencia significativa de Java.

Como respuesta a éstas y otras necesidades, Microsoft desarrolló C#. Fue creado en los laboratorios de Microsoft a finales de la década de 1990 como parte de su estrategia global .NET. La primera versión alfa fue lanzada a mediados de 2000 y el arquitecto encargado de su desarrollo fue Anders Hejlsberg.

C# está directamente emparentado con C, C++ y Java, lo cual no es accidental. Éstos son tres de los lenguajes de cómputo más utilizados (y apreciados) en el mundo. Más aún, casi todos los programadores profesionales de la actualidad conocen C, C++ o Java. Al estar construido sobre una base sólida y bien fundamentada, C# ofrece una migración sencilla desde aquellos lenguajes. Como no era necesario ni deseable que Hejlsberg “reinventara la rueda”, tuvo la libertad de enfocarse en las mejoras e innovaciones de C#.

El árbol familiar de C# se muestra en la figura 1-1. El abuelo de C# es C. A partir de éste, C++ derivó su sintaxis, muchas de sus palabras clave y operadores. C# se construyó y mejoró el modelo de objetos definido por C++. Si conoces C o C++ te sentirás como en casa con C#.



**Figura 1-1** El árbol genealógico de C#

## 6 Fundamentos de C# 3.0

---

C# y Java tienen una relación un poco más compleja. Como explicamos, Java también es descendiente de C y C++. También comparte la sintaxis y el modelo de objetos de ambos lenguajes. Como Java, C# también está diseñado para producir código portable y sus programas se ejecutan en un ambiente controlado seguro. Sin embargo, C# no es descendiente de Java. En lugar de eso C# y Java son como primos, tienen ancestros en común, pero se diferencian en muchos aspectos importantes. Sin embargo, las buenas noticias son que si conoces Java, muchos de los conceptos de C# te serán familiares. De manera inversa, si en el futuro necesitas aprender Java, mucho de lo que aprendas de C# te será de utilidad.

C# contiene muchas características innovadoras que serán examinadas con detalle a lo largo de este libro, pero algunas de las más importantes se relacionan con su soporte integrado para componentes de software. De hecho, C# ha sido caracterizado como un lenguaje orientado hacia los componentes porque contiene soporte integral para la escritura de dichos componentes. Por ejemplo, C# incluye características que soportan directamente elementos constituyentes de los componentes, como propiedades, métodos y eventos. Sin embargo, la capacidad de C# para trabajar en un ambiente seguro de lenguajes cruzados, es quizás su característica más importante en la orientación a componentes.

## La evolución de C#

Desde su lanzamiento original, C# ha evolucionado rápidamente. Poco después del lanzamiento de la versión 1.0 Microsoft lanzó la versión 1.1, que contenía muchos ajustes menores y a la que no se le añadieron mayores características. Sin embargo, la situación fue muy diferente cuando se lanzó la versión 2.0. Fue un hito en el ciclo de vida de C# porque se le añadieron muchas características nuevas, como los genéricos, tipos parciales y métodos anónimos que fundamentalmente expanden los alcances, el poder y el rango del lenguaje. La versión 2.0 colocó firmemente a C# a la cabeza en el desarrollo de los lenguajes de programación. También mostró el compromiso a largo plazo de Microsoft con el lenguaje.

El siguiente lanzamiento importante de C# fue la versión 3.0, la más actualizada en el momento de escribir este libro. Dadas las muchas características añadidas en la versión 2.0, uno podría haber esperado que el desarrollo de C# se frenara un poco, para permitir que los programadores se pusieran al corriente, pero no fue así. Con la versión 3.0 Microsoft colocó de nuevo a C# a la cabeza en el diseño de lenguajes, esta vez añadiendo un conjunto de características innovadoras que redefinieron el panorama de la programación.

Tal vez las dos características más excitantes de C# 3.0 son el lenguaje integrado query (LINQ) y las expresiones lambda. LINQ te permite escribir sentencias de ejecución para bases de datos (queries) utilizando elementos de programación de C#. Las expresiones lambda son utilizadas por lo regular dentro de las expresiones LINQ. Juntas añaden una dimensión completamente nueva a la programación en C#. Otras innovaciones incluyen tipos de variable y métodos de extensión implícitos. Como este libro se basa en la versión 3.0 de C# todos estos temas son abordados.

## Cómo se relaciona C# con .NET Framework

Aunque C# es un lenguaje de computación que puede estudiarse en sí mismo, tiene relación especial con su ambiente de ejecución: .NET Framework. Existen dos razones principales. Primera,

C# fue diseñado inicialmente por Microsoft para crear código para .NET Framework. Segunda, las bibliotecas que utiliza C# son aquellas definidas por .NET Framework. De esta manera, aunque es posible separar C# del ambiente .NET, ambos están estrechamente unidos. Por estas razones, es importante tener una comprensión general de .NET Framework y por qué es importante para C#.

## ¿Qué es .NET Framework?

En pocas palabras, .NET Framework define un ambiente que soporta el desarrollo y la ejecución de aplicaciones altamente distribuibles basadas en componentes. Permite que diferentes lenguajes de cómputo trabajen juntos y proporciona seguridad, portabilidad y un modelo común de programación para la plataforma Windows.

En lo que respecta a C#, .NET Framework define dos entidades de gran importancia. La primera es el *Lenguaje Común de Tiempo de Ejecución*, que es el sistema que administra la ejecución del programa. Entre otros beneficios, el lenguaje común de tiempo de ejecución es la parte de .NET Framework que permite que los programas sean portables, soporta la programación de lenguaje cruzado y proporciona las características de seguridad.

La segunda entidad es la *biblioteca de clases* .NET. Esta biblioteca brinda acceso al ambiente de tiempo de ejecución. Por ejemplo, si quieras realizar una acción I/O, como desplegar algo en el monitor, tienes que utilizar la biblioteca de clases .NET para hacerlo. Si eres novato en la programación, el término *clase* puede ser nuevo para ti. Aunque será explicado con detalle más adelante, diremos brevemente que clase es una construcción orientada a objetos que te ayuda a organizar los programas. Mientras tus programas se apeguen a las características definidas por esta biblioteca de clases, podrán ejecutarse en cualquier computadora que soporte el tiempo de ejecución .NET. Como C# utiliza automáticamente esta biblioteca de clases, todos los programas escritos con este lenguaje son automáticamente portables para todos los ambientes que soporten .NET.

## Cómo funciona el lenguaje común de tiempo de ejecución

El lenguaje común de tiempo de ejecución (CLR, por sus siglas en inglés) maneja la ejecución del código .NET. He aquí cómo funciona: cuando compilas un programa en C#, el resultado no es un código ejecutable. En vez de ello, es un archivo que contiene un tipo especial de pseudocódigo llamado *Lenguaje Intermediario Microsoft* o MSIL por sus siglas en inglés. MSIL define un conjunto de instrucciones portables que son independientes de cualquier CPU. En esencia, MSIL define un lenguaje ensamblador portable. Por otra parte, aunque MSIL es similar conceptualmente al bytecode de Java, no son lo mismo.

El trabajo de CLR es traducir el código intermediario en código ejecutable cuando se ejecuta un programa. De esta manera, cualquier programa compilado en MSIL puede ser ejecutado en cualquier ambiente donde esté implementado CLR. Así es como, en parte, .NET Framework adquiere su portabilidad.

El Lenguaje Intermediario de Microsoft se convierte en código ejecutable utilizando el compilador *JIT*. JIT son las siglas de “just in time” o “justo a tiempo”. El proceso funciona de la siguiente manera: cuando se ejecuta un programa .NET, CLR activa el compilador JIT. Este último convierte MSIL en código nativo de la computadora cada parte del programa conforme se vaya

necesitando. De esta manera, tu programa escrito en C# de hecho se ejecuta como código nativo de la computadora, aunque inicialmente se haya compilado en MSIL. Esto significa que tu programa se ejecuta casi tan rápidamente como si hubiera sido compilado en código nativo desde el inicio, pero gana los beneficios de la portabilidad y la seguridad de MSIL.

Además de MSIL, otro elemento se añade a los datos de salida cuando compilas un programa C#: *metadatos*. Los metadatos son aquellos que utiliza tu código para interactuar con otros códigos. Los metadatos están contenidos en el mismo archivo que MSIL.

Por fortuna, para los propósitos de este libro, y para la mayoría de las tareas de programación, no es necesario que conozcas a fondo CLR, MSIL y los metadatos. C# se encarga de manejarlos por ti.

### Código controlado vs. código sin control

En general, cuando escribes un programa en C# estás creando lo que se conoce como *código controlado*. Este código se ejecuta bajo el control del lenguaje común de tiempo de ejecución, como se describió anteriormente. Como se ejecuta bajo la supervisión de CLR, el código controlado está sujeto a ciertas restricciones y también aporta ciertos beneficios. Las restricciones son fáciles de describir y entender: el compilador debe producir un archivo MSIL dirigido hacia el CLR (de lo cual se encarga C#) y utiliza la biblioteca .NET Framework (tarea de C# también). Los beneficios del código controlado son muchos, incluyendo la administración de la memoria moderna, la posibilidad de mezclar lenguajes, mejor seguridad, soporte para el control de versiones y una interacción limpia de los componentes de software.

Lo contrario del código controlado es el *código sin control*. Este último no se ejecuta bajo el lenguaje común de tiempo de ejecución. Por lo mismo, todos los programas de Windows anteriores a la creación de .NET Framework utilizan código sin control. Es posible que ambos códigos funcionen en conjunto, por lo que el hecho de que C# genere código controlado no restringe su capacidad para operar en conjunto con otros programas preexistentes.

### Pregunta al experto

**P:** Para cumplir con las tareas de portabilidad, seguridad y programación de lenguajes mixtos, ¿por qué es necesario crear un nuevo lenguaje como C#? ¿No sería posible adaptar algún otro lenguaje como C++ para dar soporte a .NET Framework?

**R:** Sí, es posible adaptar C++ para que produzca código compatible con .NET Framework que se ejecute bajo el CLR. De hecho, Microsoft añadió lo que se conoce como *extensiones de administración a C++*. Sin embargo, este acercamiento ha quedado obsoleto y ha sido reemplazado por un conjunto de palabras clave extendidas y sintaxis definida por el estándar Ecma C++/CLI (CLI son las siglas en inglés de Infraestructura de Lenguaje Común). Aunque C++/CLI hace posible exportar código existente a .NET Framework, el nuevo desarrollo en este ambiente es mucho más sencillo con C# porque se diseñó originalmente pensando en él.

## La especificación del lenguaje común

Aunque todo el código controlado obtiene el beneficio que proporciona el CLR, si tu código va a ser utilizado por otros programas escritos en diferentes lenguajes, para obtener el mayor provecho, debe ceñirse a las especificaciones del lenguaje común (CLS, por sus siglas en inglés). El CLS describe un conjunto de características, como los tipos de datos, que tienen en común diversos lenguajes de programación. La adaptabilidad del CLS es en especial importante cuando se crean componentes de software que serán usados por otros lenguajes. Aunque no debemos preocuparnos por el CLS dados los propósitos de este libro, es un tema que necesitarás abordar cuando comiences a escribir programas comerciales.

## Programación orientada a objetos

En el centro de C# está la programación orientada a objetos (POO). La metodología orientada a objetos es inseparable de C# y todos los programas escritos en este lenguaje, o por lo menos la mayoría, son orientados a objetos. Dada su enorme importancia para C#, es importante comprender los principios básicos de la POO antes de que escribas incluso un sencillo programa en C#.

La POO es un poderoso acercamiento al trabajo de la programación. Las metodologías de programación han cambiado drásticamente desde la invención de la computadora, principalmente para acomodar la complejidad creciente de los programas. Por ejemplo, cuando fueron inventadas las computadoras, la programación se realizaba introduciendo las instrucciones binarias del código de máquina utilizando el panel frontal de la computadora. El método funcionaba mientras los programas estuvieran construidos por unos centenares de instrucciones. Cuando los programas se hicieron más grandes, se inventó el lenguaje ensamblador para que el programador pudiera manejar la magnitud y complejidad de los nuevos programas utilizando representaciones simbólicas del código de máquina. Cuando los programas crecieron aún más, se crearon lenguajes de alto nivel como FORTRAN y COBOL con el fin de ofrecer una serie de herramientas que hicieran posible para el programador manejar la creciente complejidad. Cuando estos programas comenzaron a alcanzar su límite, se creó la programación estructurada.

Considera lo siguiente: en cada punto de quiebre en el desarrollo de la programación, se han creado técnicas y herramientas nuevas para permitir que los programadores manejaran la creciente complejidad de los programas. En cada nuevo paso, el nuevo enfoque tomó los mejores elementos de los métodos anteriores y los llevó un paso adelante. Lo mismo se aplica a la programación orientada a objetos. Antes de la POO muchos proyectos estuvieron cerca o rebasaron el punto donde la programación estructurada ya no funcionaba. Se necesitaba un nuevo medio para manejar la complejidad, y la programación orientada a objetos fue la solución.

La programación orientada a objetos tomó las mejores ideas de la programación estructurada y las combinó con varios conceptos novedosos. El resultado fue una manera diferente y mejor de organizar un programa. En el sentido más general, los programas pueden organizarse en una de dos maneras: alrededor de su código (lo que sucede) o alrededor de sus datos (lo que es afectado). Al utilizar sólo técnicas de lenguaje estructurado, por lo general los programas se organizan alrededor del código. Este enfoque puede pensarse como “código que actúa sobre los datos”.

Los programas orientados a objetos funcionan de manera inversa. Están organizados alrededor de los datos, cuyo principio clave es “datos controlando acceso al código”. En un lenguaje orientado

a objetos, uno define los datos y las rutinas que tienen permiso para actuar sobre ellos. De esta manera, los tipos de datos definen con precisión qué tipo de operaciones pueden ser aplicadas a los datos.

Para soportar los principios de la programación orientada a objetos, todos los lenguajes que tienen este enfoque, incluido C#, tienen tres aspectos en común: encapsulado, polimorfismo y herencia. Examinemos cada uno de ellos.

### Encapsulado

El encapsulado es un mecanismo de programación que une el código y los datos que éste manipula, manteniendo a ambos seguros de interferencias exteriores y usos erróneos. En un lenguaje orientado a objetos, el código y los datos pueden conjuntarse de tal manera que se crea una *caja negra* autocontenido. Dentro de la caja está todo el código y los datos necesarios. Cuando tu código y datos están unidos de esta manera, se crea un objeto. En otras palabras, un objeto es un dispositivo que soporta el encapsulado.

Dentro de un objeto, el código, los datos o ambos, pueden ser *privados* o exclusivos de ese objeto, o bien pueden ser *públicos*. El código privado es conocido y accesible exclusivamente por las diferentes partes que conforman el objeto. Esto es, el código o los datos privados no pueden ser accesados por una pieza del programa que exista fuera del objeto del que forman parte. Cuando el código o los datos son públicos, otras partes del programa pueden tener acceso a ellos, aunque estén definidos dentro del objeto en cuestión. Por lo regular, las partes públicas de un objeto son utilizadas para proporcionar una interfaz controlada a los elementos privados del mismo.

La unidad básica de C# para el encapsulado es la *clase*. Una clase define la forma de un objeto. Define tanto los datos como el código que operará sobre ellos. C# utiliza una especificación de clase para construir *objetos*. Los objetos son instancias de la clase. De esta manera, una clase es esencialmente un conjunto de planos que especifican cómo construir un objeto.

El código y los datos que constituyen una clase son llamados *miembros* de la clase. Específicamente, los datos definidos por la clase son conocidos como *variables miembro* o *variables de instancia*. El código que opera sobre esos datos es conocido como *métodos miembro* o simplemente *métodos*. “Método” es el término de C# para las subrutinas. Si estás familiarizado con C/C++ tal vez te sea de utilidad saber que aquello que un programador de C# llama *método*, un programador de C/C++ lo llama *función*. Como C# es descendiente directo de C++, en algunas ocasiones se utiliza también el término “función” cuando se hace referencia a un método de C#.

### Polimorfismo

El polimorfismo (palabra proveniente del griego y que significa “muchas formas”) es la cualidad que le permite a una interfaz acceder a una clase general de acciones. Un ejemplo sencillo de polimorfismo se encuentra en el volante de un automóvil. El volante (la interfaz) es siempre el mismo, no importa qué tipo de mecanismo utilice, es decir, el volante funciona de la misma manera si el auto es de dirección mecánica, hidráulica o algún otro mecanismo de dirección. De esta manera, girar el volante hacia la izquierda hará que el automóvil vire hacia la izquierda, sin importar el tipo de dirección que utiliza. El beneficio de las interfaces comunes es, por supuesto, que una vez que aprendes a utilizar un volante puedes dirigir cualquier tipo de automóvil, porque siempre es lo mismo.

El mismo principio puede aplicarse a la programación. Por ejemplo, considera un orden en pila (lo primero que entra es lo último que sale). Puede ser que tengas un programa que requiera

tres tipos diferentes de pilas: uno para organizar valores enteros, otro para valores de punto flotante y otro más para caracteres. En este caso, el algoritmo que aplica el orden en pila es el mismo, aunque los datos que serán almacenados sean diferentes. En un lenguaje diferente al orientado a objetos necesitarías crear tres tipos diferentes de rutinas de pila, donde cada conjunto utiliza nombres distintos. Sin embargo, gracias al polimorfismo, en C# puedes especificar la forma general de la pila una sola vez y después utilizarla para tres situaciones específicas. De esta manera, si sabes utilizar una pila, puedes utilizarlas todas.

De manera más general, el concepto del polimorfismo es expresado a menudo con la frase “una interfaz, múltiples métodos”. Esto significa que es posible diseñar una interfaz genérica para un grupo de actividades similares. El polimorfismo ayuda a reducir el grado de complejidad permitiendo que la misma interfaz sea utilizada para especificar una *clase general de acción*. Es trabajo del compilador seleccionar la *acción específica* (es decir, el método) que debe aplicar en cada situación. El programador no necesita hacer esta selección manualmente. Sólo debe recordar y utilizar la interfaz genérica.

## Herencia

La herencia es el proceso mediante el cual un objeto puede adquirir las propiedades de otro. Esto es importante porque da soporte al concepto de clasificación jerárquica. Si lo piensas, la mayor parte del conocimiento es manejable por una clasificación jerárquica (de arriba abajo). Por ejemplo, una manzana Roja Deliciosa es parte de la clasificación *manzana*, que a su vez es parte de la clase *fruta*, que se localiza dentro de una clase mayor llamada *comida*. Esto es, la clase *comida* posee ciertas cualidades (comestible, nutritiva y demás) que por lógica se aplican a su subclase *fruta*. Además de tales cualidades, la clase *fruta* tiene características específicas (jugosidad, dulzura y demás) que la distinguen de otra comida. La clase *manzana* define aquellas cualidades específicas de una manzana (crece en árboles, no se da en climas tropicales, etc.). Una manzana Roja Deliciosa heredaría, a su vez, todas las cualidades de todas las clases precedentes y definiría sólo aquellas cualidades que la hacen única.

## Pregunta al experto

**P:** Ha dicho que la programación orientada a objetos es una manera efectiva de manejar programas grandes. Sin embargo, parece que la POO puede saturar programas pequeños. Dice que todos los programas escritos en C# son, de alguna manera, orientados a objetos, ¿esto significa que no es posible hacer programas pequeños con ese lenguaje?

**R:** No. Como verás, para programas pequeños las características orientadas a objetos de C# son casi transparentes. Aunque es cierto que C# se ciñe estrictamente al modelo de objetos, tú tienes toda la libertad para decidir hasta qué punto emplearla. Para programas pequeños, la orientación a objetos es apenas perceptible. Mientras tu programa crezca, integrarás más características orientadas a objetos sin esfuerzo.

Sin el uso de jerarquías, cada objeto tendría que definir de manera explícita todas sus características. Con el uso de la herencia, un objeto sólo necesita definir aquellas cualidades que lo hacen único dentro de su clase. Puede heredar sus atributos generales de sus padres. De esta manera, es el mecanismo de la herencia lo que hace posible que un objeto sea una instancia específica de un caso mucho más general.

## Crear, compilar y ejecutar tu primer programa en C#

Es hora de aprender a crear, compilar y ejecutar un programa en C#. Como ésta es una guía de “manos a la obra” para la programación en C#, completar exitosamente esta tarea es un primer paso necesario. He aquí el programa que utilizaremos:

```
/*
Éste es un programa sencillo en C#

Llama a este programa Ejemplo.cs.

*/
using System;

class Example{

    // Un programa C# principia con una invocación Main().
    static void Main(){
        Console.WriteLine("C# te da el poder de la programación.");
    }
}
```

Aunque corto, este programa contiene varias características clave que son comunes a todos los programas de C#. Una vez que hayas aprendido a compilarlo y ejecutarlo, lo examinaremos en detalle.

## Obtener un compilador C# 3.0

Para crear, compilar y ejecutar un programa C#, necesitarás una copia de Microsoft Visual C#. Este libro utiliza Visual C# 2008, que es el compilador que soporta la versión 3.0 de C#. Aunque muchos de los programas presentados en este libro pueden ser compilados por versiones anteriores del mismo lenguaje, necesitarás Visual C# 2008 para aprender a manejar las características más recientes.

Si en este momento no cuentas con Visual C# 2008, es necesario que lo adquieras. Microsoft lo proporciona de diferentes formas, incluyendo una versión comercial que puede ser comprada. Sin embargo, en el momento en que este libro está en proceso, también puedes adquirir una copia gratuita descargando la edición Express. La edición Express de C# contiene un compilador completamente funcional que soporta todas las características de la versión 3.0 y por tanto es capaz de compilar todos los ejemplos de este libro. También incluye Visual Studio, que es el ambiente

de desarrollo integrado de Microsoft (IDE, por sus siglas en inglés). Aunque la versión Express no contiene todas las herramientas deseables para un desarrollador profesional, es perfecto para aprender C#. En el momento en que se escribe este libro, Visual C# 2008 Express puede ser descargado de [www.microsoft.com/express](http://www.microsoft.com/express). Todo el código que aparece en este libro ha sido probado con este compilador.

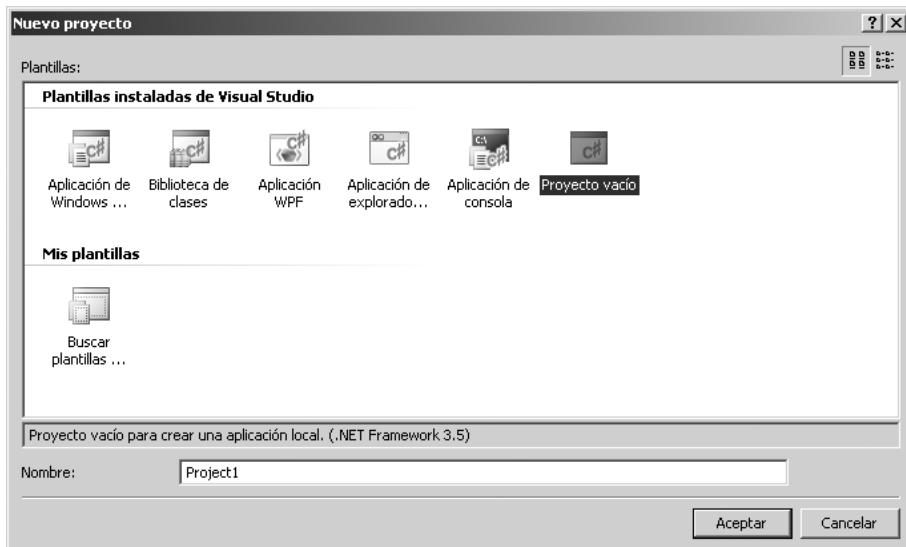
Al usar Visual C#, hay dos enfoques generales con los que puedes crear, compilar y ejecutar un programa de C#. En primer lugar, puedes usar Visual Studio ICE; en segundo lugar, puedes usar el compilador de línea de comandos, **csc.exe**. Ambos métodos se describen aquí.

## Utilizar el IDE de Visual Studio

Como mencionamos, Visual Studio es el ambiente de desarrollo integrado de Microsoft. Te permite editar, compilar, ejecutar y depurar un programa de C#, todo ello sin tener que abandonar el bien desarrollado ambiente. Visual Studio ofrece un medio conveniente y de gran ayuda para administrar tus programas. Es mucho más efectivo para programas grandes, pero puede ser utilizado con gran éxito en programas pequeños, como los que constituyen los ejemplos de este libro.

Los pasos necesarios para editar, compilar y ejecutar un programa C# utilizando la IDE de Visual Studio 2008 se muestran aquí. Estos pasos dan por hecho que estás utilizando el IDE proporcionado por Visual Studio C# Express 2008. Pueden existir pequeñas diferencias con otras versiones de Visual Studio 2008.

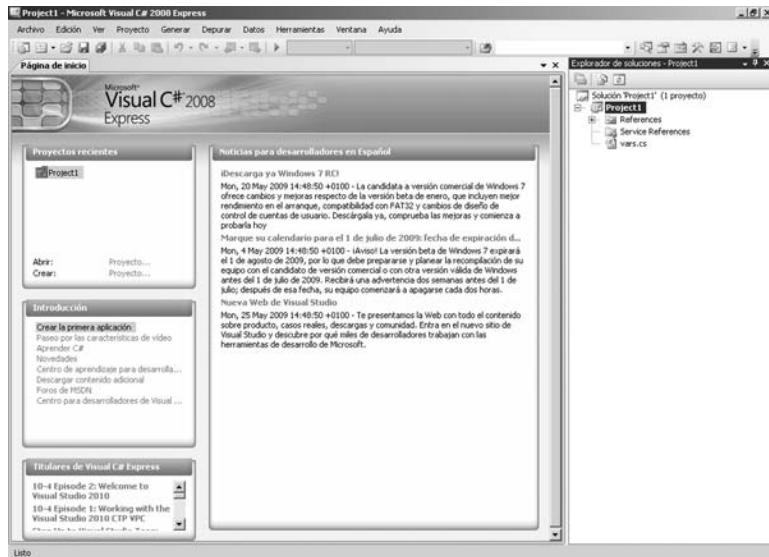
1. Crea un nuevo proyecto vacío C# seleccionando Archivo | Nuevo Proyecto. A continuación, selecciona Proyecto Vacío.



Haz clic en Aceptar para crear el proyecto.

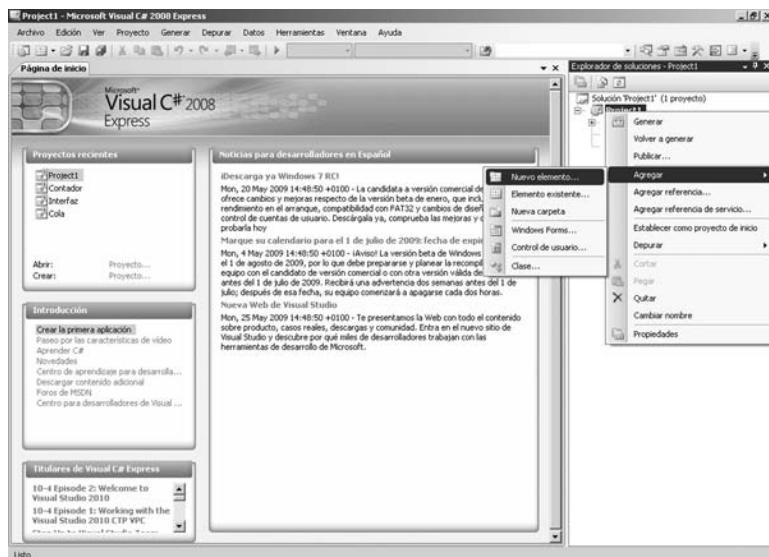
## 14 Fundamentos de C# 3.0

2. Una vez que el proyecto ha sido creado, el IDE de Visual Studio se verá así:

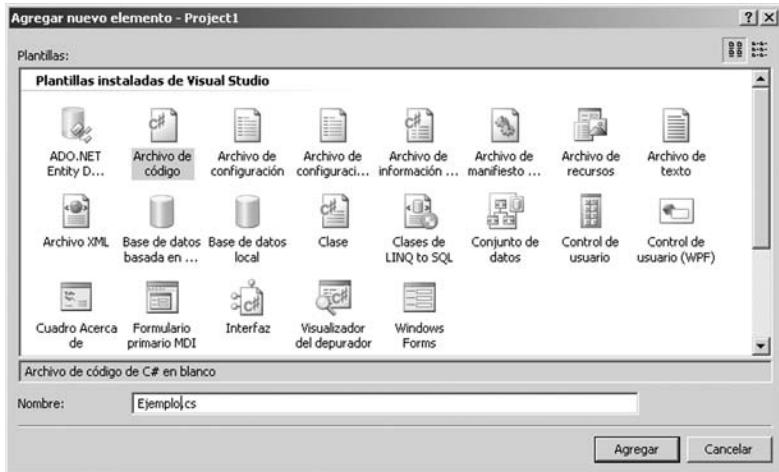


Si por alguna razón no ves la ventana del Explorador de soluciones, activala seleccionando Explorador de soluciones en el menú Ver.

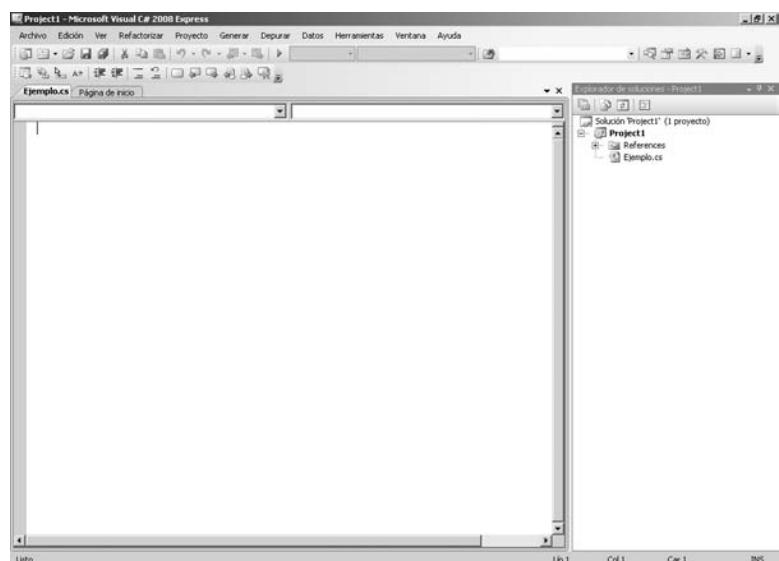
3. En este punto el proyecto está vacío y necesitas añadirle código C#. Para hacerlo, haz clic con el botón derecho en Project 1 en el Explorador de soluciones y selecciona Agregar. Verás lo siguiente:



4. A continuación, selecciona Nuevo elemento. Esto provoca que aparezca la ventana de diálogo Añadir nuevo elemento. Selecciona Archivo de código y cambia el nombre por **Ejemplo.cs**, como se muestra aquí:



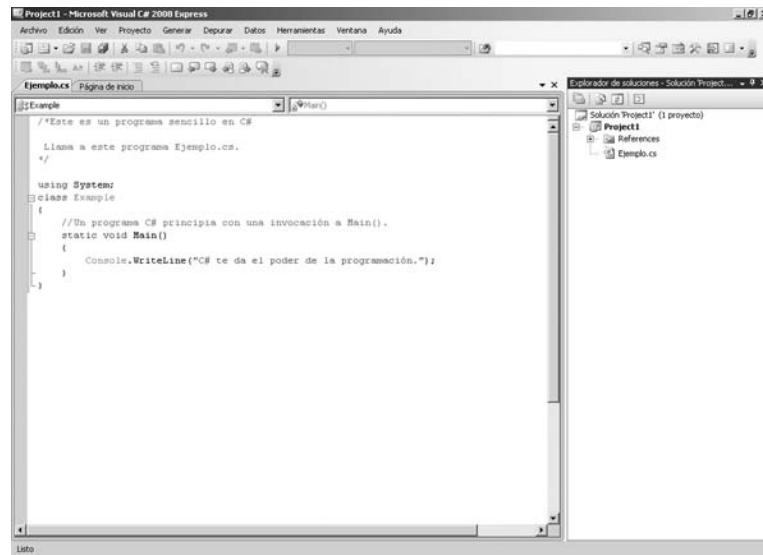
5. A continuación, añade el archivo al proyecto haciendo clic en Agregar. Ahora tu pantalla se verá así:



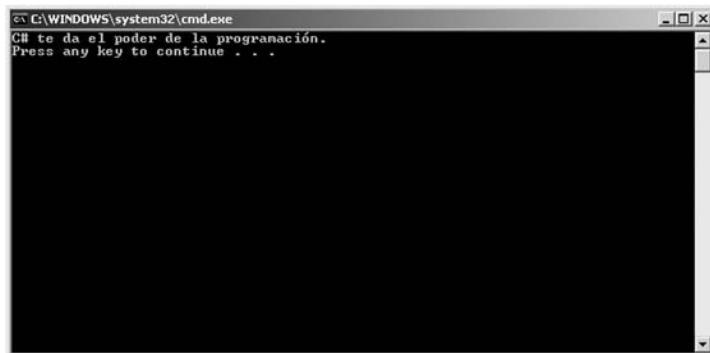
6. A continuación, escribe el programa ejemplo en la ventana Ejemplo.cs y guarda el archivo. (Puedes descargar el código fuente de los programas que aparecen en este libro de

## 16 Fundamentos de C# 3.0

<http://www.mcgraw-hill-educacion.com> [busca por ISBN] para que no tengas que escribirlos manualmente.) Cuando finalices, tu pantalla se verá así:



7. Compila el programa seleccionando Generar solución del menú Generar.
8. Ejecuta el programa seleccionando Iniciar sin depurar del menú Depurar. Cuando ejecutas el programa, verás la ventana que aparece a continuación:



Como lo muestran las instrucciones anteriores, compilar pequeños programas de ejemplo utilizando el IDE requiere seguir ciertos pasos. Sin embargo, no necesitas crear un nuevo proyecto para cada ejemplo que aparece en este libro. En vez de ello, puedes utilizar el mismo proyecto C#. Simplemente elimina el archivo fuente actual y añade uno nuevo. Luego compílalo y ejecútalo. Con este enfoque se simplifica considerablemente el proceso. Debes comprender, sin embargo, que en las aplicaciones del mundo real cada programa utilizará su propio proyecto.

**TIP**

Aunque las instrucciones anteriores son suficientes para compilar y ejecutar los programas de este libro, si planeas utilizar el IDE de Visual Studio como tu ambiente de trabajo principal, es necesario que te familiarices con todas sus capacidades y características. Se trata de un ambiente de desarrollo muy poderoso que te ayuda a administrar proyectos grandes. También proporciona una manera de organizar los archivos y recursos asociados con el proyecto. Vale la pena el tiempo y esfuerzo que destines para aprender a manejar Visual Studio.

## Usar csc.exe, el compilador de línea de comandos de C#

Aunque es probable que utilices el IDE de Visual Studio para tus proyectos comerciales, algunos lectores encontrarán la línea de comandos más conveniente para compilar, especialmente para compilar y ejecutar los programas ejemplo que aparecen en este libro. La razón es que no necesitas crear un proyecto para el programa. Simplemente puedes crear el programa, compilarlo y ejecutarlo, todo desde la línea de comandos. Por tanto, si sabes utilizar la ventana de comandos y la interfaz basada en caracteres, usar el compilador de línea de comandos será más fácil y rápido en comparación con el IDE.

**PRECAUCIÓN**

Si no estás familiarizado con la consola de comandos, quizás sea más recomendable utilizar el IDE de Visual Studio. Aunque sus comandos no son difíciles, intentar aprenderlos a la par con las instrucciones de C# representaría una experiencia retadora.

Para crear y ejecutar programas utilizando el compilador de línea de comando C#, sigue los siguientes pasos:

- 1.** Captura el código utilizando un editor de texto.
- 2.** Compila el programa con el uso de **csc.exe**.
- 3.** Ejecuta el programa.

### Capturar el programa

Como se mencionó, los programas que aparecen en este libro están disponibles en el sitio Web de McGraw-Hill: <http://www.mcgraw-hill-educacion.com>. Aunque si quieres capturar los programas a mano, puedes hacerlo. En este caso, debes capturar el código utilizando un editor de texto, como la libreta de notas (notepad). Recuerda que debes crear archivos sólo texto y no archivos con formato como los de procesadores de palabras, porque la información del formato que añaden estos últimos puede confundir al compilador de C#. Cuando hayas terminado de capturar el programa, debes nombrarlo **Ejemplo.cs**.

### Compilar el programa

Para compilar el programa, ejecuta el compilador de C# **csc.exe**, especificando el nombre del archivo fuente en la línea de comandos, como se muestra a continuación:

C:\>csc Ejemplo.cs

El compilador **csc** crea un archivo llamado **Ejemplo.exe** que contiene la versión MSIL del programa. Aunque MSIL no es código ejecutable, de cualquier manera contiene un archivo **exe**. El lenguaje común de tiempo de ejecución automáticamente invoca el compilador JIT cuando se intenta ejecutar el programa **Ejemplo.exe**. Pero ten presente que si intentas ejecutar éste o cualquier otro programa **exe** que contenga MSIL en una computadora en la que no está instalado .NET Framework, el programa no se ejecutará porque no encontrará el CLR.

### **NOTA**

Antes de ejecutar **csc.exe** necesitas abrir la Consola de Comando que está configurada para Visual Studio. La manera más sencilla de hacerlo es seleccionar "Consola de Visual Studio 2008" desde el menú Inicio, Herramientas de Visual Studio. Como alternativa, puedes iniciar una consola de comandos sin configurar y luego ejecutar el archivo por lotes **vsvars32.bat**, que proporciona Visual Studio. Sin embargo, es posible que tengas problemas con el método de la línea de comandos. En el momento en que escribo este libro, la edición Express de Visual C# 2008 no proporciona el menú Herramientas de Visual Studio ni el archivo por lotes **vsvars32.bat**. Por lo mismo, si estás utilizando la versión Express, no podrás configurar automáticamente una consola. En ese caso utiliza el IDE de Visual Studio. Aunque la versión Express de Visual C++ 2008 sí contiene tanto el archivo por lotes **vsvars32.bat** como la opción para seleccionar la consola de Visual Studio. Así, en caso de que instales también la versión Express de C++ podrás configurar apropiadamente la consola de comandos que también funciona para C#.

### **Ejecutar el programa**

Para ejecutar el programa simplemente escribe su nombre en la línea de comandos, como se muestra aquí:

```
C: \>Ejemplo
```

Cuando se ejecuta el programa, aparecen los siguientes datos de salida:

```
C# te da el poder de la programación.
```

## **El primer programa ejemplo línea por línea**

Aunque **Ejemplo.cs** es muy corto, incluye muchas características clave que son comunes a todos los programas C#. Como éste es tu primer programa, merece un examen profundo. Comenzaremos por su nombre.

El nombre de un programa C# puede seleccionarse de manera arbitraria. Contrariamente a algunos programas de cómputo (en particular Java) en los que el nombre del archivo del programa es muy importante, no sucede lo mismo con C#. Te indicamos que nombraras el primer programa **Ejemplo.cs** para poder aplicar las instrucciones de compilación y ejecución, pero en lo que concierne a C# podrías haberlo llamado de cualquier otra manera. Por ejemplo, pudiste llamarlo **Muestra.cs**, **Test.cs** e incluso **MiPrograma.cs**.

Por convención, los programas C# utilizan la extensión de archivo **.cs**, y es una convención que debes seguir. De la misma manera, muchos programadores ponen al archivo el mismo nombre de la clase principal ahí definida. Por esa razón el archivo **Ejemplo.cs** recibió tal nombre. Como los nombres de los archivos en C# son arbitrarios, la mayoría de los programas de ejemplo que aparecen en este libro no tendrán un nombre específico; puedes darles el nombre que gustes.

El programa comienza con las siguientes líneas:

```
/*
Éste es un programa sencillo en C#.
Llama a este programa Ejemplo.cs.
*/
```

Éste es un *comentario*. Como la mayoría de los lenguajes de programación, C# te permite introducir anotaciones dentro del código fuente del programa. El contenido de un comentario es ignorado por el compilador; su objetivo es describir o explicar la operación del programa a quien esté leyendo el código fuente. En este caso, el comentario describe el programa y te recuerda que debes nombrar **Ejemplo.cs** al archivo que lo contiene. Por supuesto, en aplicaciones reales los comentarios explican, por lo regular, el funcionamiento de cierta parte del programa o alguna característica específica que realiza.

C# permite tres estilos de comentarios. El que aparece al principio del programa ejemplo se llama *comentario multilínea*. Este tipo de comentario debe comenzar con /\* y terminar con \*/. Cualquier elemento dentro de estos dos símbolos es ignorado por el compilador. Como su nombre lo sugiere, este tipo de comentario puede contener varias líneas.

La siguiente línea del programa es

```
using System;
```

Esta línea indica que el programa está utilizando la nomenclatura **System**. En C# una *nomenclatura* define una región declarativa. Aunque más adelante estudiaremos con detalle las nomenclaturas, ahora nos es de utilidad una breve descripción. Una nomenclatura proporciona un medio de mantener un conjunto de nombres separado de otro. En esencia, los nombres declarados en una nomenclatura no entrarán en conflicto con nombres idénticos declarados en otra nomenclatura. La nomenclatura utilizada por el programa es **System**, que es la nomenclatura reservada para elementos asociados con la biblioteca de clases de .NET Framework, que es precisamente la que utiliza C#. La palabra clave **using** declara que el programa está utilizando los nombres de una nomenclatura dada.

La siguiente línea de código en el programa se muestra a continuación:

```
class Example {
```

Esta línea utiliza la palabra clave **class** para declarar que una nueva clase está siendo definida. Como se mencionó anteriormente, la clase es la unidad básica de encapsulación de C#. **Ejemplo** es el nombre de la clase. La definición de clase comienza con una llave de apertura ({) y termina con una de clausura (}). Los elementos ubicados dentro de las llaves son miembros de esa clase. Por el momento, no te preocupes por los detalles de la clase, sólo ten presente que la actividad del programa ocurre dentro de una de ellas.

La siguiente línea en el programa es un *comentario de una sola línea*, que se muestra a continuación:

```
// Un programa C# principia con una invocación a Main().
```

Éste es el segundo tipo de comentario que soporta C#. Un *comentario de una sola línea* comienza con // y termina al final de la línea. Como regla general, los programadores utilizan comentarios

multilínea para hacer anotaciones largas y los comentarios de una sola línea para observaciones breves o descripciones que no requieran más de una línea.

La siguiente línea de código es

```
static void Main() {
```

Esta línea da inicio al método **Main()**. Como se mencionó anteriormente, en C# las subrutinas reciben el nombre de método. Como decíamos, ésta es la línea con la cual el programa comenzará a ejecutarse. Todas las aplicaciones C# comienzan a ejecutarse invocando el método **Main()**. El significado completo de cada parte de esta línea no puede ser explicada en este momento, porque implica la comprensión detallada de otras características de C#. Sin embargo, como muchos ejemplos en este libro utilizarán esta línea de código, se justifica una breve explicación.

La línea comienza con la palabra clave **static**. Un método que es modificado por **static** puede ser invocado antes de que un objeto de su clase sea creado. Esto es necesario porque **Main()** es invocado cuando arranca el programa. La palabra clave **void** indica que **Main()** no regresa ningún valor. Como verás, los métodos también pueden regresar valores. Los paréntesis vacíos que siguen a **Main** indican que no se le transmite ninguna información al método. Como también verás, es posible transmitir información a cualquier método, incluyendo **Main()**. El último carácter en la línea es la llave de apertura ({}, que señala el comienzo del cuerpo de **Main()**. Todo el código que ejecuta un método estará encerrado entre la llave de apertura y la de clausura.

La siguiente línea de código se muestra a continuación. Advierte que se localiza dentro de **Main()**.

```
Console.WriteLine("C# te da el poder de la programación.");
```

Esta línea envía la cadena de salida “C# te da el poder de la programación.” a la pantalla, seguida de una nueva línea en blanco. Los datos de salida son manejados, de hecho, por el método integrado **WriteLine()**. En este caso **WriteLine()** muestra la cadena de caracteres que le es transmitida. La información que es transmitida a un método recibe el nombre de *argumento*. Además de cadenas de caracteres, **WriteLine()** puede ser utilizada para mostrar otro tipo de información, como datos numéricos. La línea comienza con la palabra **Console**, que es el nombre de una clase predefinida que soporta datos de entrada y salida (I/O) en la consola. Al conectar **Console** con **WriteLine()**, le estás diciendo al compilador que **WriteLine()** es un miembro de la clase **Console**. El hecho de que C# utilice un objeto para definir los datos de salida de la consola es una clara evidencia de su naturaleza orientada a objetos.

### Pregunta al experto

**P:** Dijo que C# permite tres tipos de comentarios, pero sólo mencionó dos. ¿Cuál es el tercero?

**R:** El tercer tipo de comentario permitido por C# es un *comentario de documentación*, también llamado *comentario XML*. Un comentario de documentación utiliza etiquetas XML para ayudarte a crear código autodocumentado.

Advierte que la declaración **WriteLine()** termina con un punto y coma (;), lo mismo que la declaración **using System** anteriormente pero en el mismo programa. En general, todas las declaraciones de C# terminan con un punto y coma. La excepción a esta regla es el *bloque*, que comienza con { y termina con }. Ningún bloque termina con punto y coma; por eso las líneas terminan con } y no con punto y coma. Los bloques proporcionan un mecanismo para agrupar declaraciones y son analizados con detalle más tarde en este capítulo.

La primera llave de clausura ()} indica el final del método **Main()**, y la última } indica la culminación de la definición de clase **Ejemplo**.

Un último punto de importancia: C# es sensible a las mayúsculas; olvidar esto puede causar serios problemas. Por ejemplo, si en forma accidental escribes **main** en lugar de **Main**, o **writeline** en vez de **WriteLine**, el programa resultante será incorrecto. Más aún, aunque el compilador de C# sí *compilará* clases que no contengan el método **Main()**, no será posible utilizar uno como punto de entrada para ejecutar el programa. Así, si escribiste incorrectamente **Main**, C# compilará tu programa, pero verás que aparece un mensaje de error indicando que el programa **Ejemplo.exe** no tiene un punto de entrada predefinido.

## Manejo de errores de sintaxis

Si aún no lo has hecho, escribe, compila y ejecuta el programa de ejemplo. Como tal vez lo sepas a partir de tu anterior experiencia en la programación, es muy fácil escribir accidentalmente algo incorrecto cuando se escribe el código fuente. Si esto sucede, el compilador reportará un *error de sintaxis* cuando intente compilar el programa. El mensaje mostrado contendrá el número de la línea y la posición del carácter donde fue localizado el error, así como una descripción del mismo.

Aunque los errores de sintaxis reportados por el compilador son, obviamente, de gran utilidad, también es posible que no sean del todo certeros. El compilador de C# intenta darle sentido a tu código fuente sin importar lo que hayas escrito. Por esta razón, el error reportado no siempre refleja la causa verdadera del problema. Por ejemplo, en el programa anterior una omisión accidental de la llave de apertura del método **Main()** generaría la siguiente secuencia de errores al compilarlo desde la línea de comandos **esc** (errores similares son generados cuando se compila utilizando IDE):

```
EX1. CS(12, 21): error CS1002: ; expected
EX1. CS(13, 22): error CS1519: Invalid token '(' in class, struct, or
interface member declaration
EX1. CS(15, 1): error CS1022: Type or namespace definition, or end-of-file
expected
```

Claramente, el primer mensaje de error es por completo erróneo, porque el signo faltante no es un punto y coma sino una llave de apertura. Los siguientes dos mensajes son igualmente confusos.

El punto de este breve análisis es que cuando tu programa contenga un error de sintaxis no tomes como verdad absoluta la información que aparece en los mensajes de error que genera el compilador, porque pueden ser poco certeros. Es probable que tengas que dar una “segunda opinión” sobre el mensaje de error para encontrar el verdadero problema. También es conveniente que revises las líneas del código inmediatas anteriores a la marcada como error, porque algunas veces un error no se reportará hasta que se hayan ejecutado ciertas acciones y es posible que la falla se localice antes del punto indicado por el compilador.

## Una pequeña variación

Aunque todos los programas que aparecen en este libro utilizarán la declaración `using System;`

al inicio, técnicamente no es necesario. Sin embargo, es una convención valiosa. La razón por lo que técnicamente no es necesario es que en C# siempre puedes *calificar completamente* un nombre con la nomenclatura a la que pertenece. Por ejemplo, la línea

```
Console.WriteLine("Un programa C# sencillito.");
```

puede escribirse como

```
System.Console.WriteLine("Un programa C# sencillito.");
```

En ocasiones verás código C# que utiliza este método, pero no es muy común. La mayoría de los programadores de C# utilizan la declaración **using System** al inicio de sus programas, como los ejemplos que aparecen en este libro. Al hacerlo se evita el tedio de tener que especificar la nomenclatura **System** cada vez que se utiliza un miembro de la misma. Sin embargo, es importante comprender que puedes calificar explícitamente un nombre dentro de una nomenclatura cuando sea necesario.

## Usar una variable

Tal vez ningún constructor sea tan importante en los lenguajes de programación como la variable. Una *variable* es una locación de memoria con un nombre determinado a la que se le puede asignar un valor. Se le llama variable porque su valor puede cambiar durante la ejecución del programa. En otras palabras, el valor de la variable es cambiante, no fijo.

Comencemos con un ejemplo. El siguiente programa crea tres variables llamadas **largo**, **ancho** y **área**. Utiliza estas variables para calcular y mostrar en pantalla el área de un rectángulo cuyas dimensiones son 9 por 7.

```
// Este programa presenta una introducción a las variables.  
using System;  
  
class UseVars {  
    static void Main() {  
        int largo; // así se declara una variable  
        int ancho; // así se declara otra variable ← Declara variables.  
        int área; // ésta es una tercera variable  
  
        // Asignar a la variable largo el valor de 9.  
        largo = 9; ← Así se asigna el valor 9 a largo.  
  
        // Esto despliega el valor actual de la variable largo.  
        Console.WriteLine("el largo es de " + largo);  
  
        // Asignar a la variable ancho el valor de 7.  
        ancho = 7; ← Así se asigna el valor 7 a ancho.
```

```
// Esto despliega el valor actual de la variable ancho.  
Console.WriteLine("el ancho es de " + ancho);  
  
// Asignar a la variable área el producto de largo por ancho.  
área = ancho * largo; ← Multiplica largo por ancho  
y asigna el resultado a área.  
  
// Desplegar el resultado  
Console.WriteLine("el área de ancho * largo: ");  
Console.WriteLine(área);  
}  
}
```

Cuando ejecutes el programa, verás los siguientes datos de salida como resultado:

```
largo: 9  
ancho: 7  
el área de ancho * largo: 63
```

Este programa introduce varios conceptos nuevos. Primero, la declaración `int largo;` // así se declara una variable expone una variable llamada **largo** de tipo entero (**int**). En C# todas las variables deben ser declaradas antes de utilizarse. De igual manera, también debe expresarse el tipo de valores que puede contener esa variable; esto se conoce como *tipo* de variable. En este caso, **largo** puede contener valores enteros, los cuales son números naturales o enteros. En C#, para declarar una variable como entero se antepone la palabra clave **int** al nombre de variable. De esta manera, la declaración anterior indica que una variable llamada **largo** es de tipo **int**.

Las siguientes dos líneas declaran otras dos variables **int**, llamadas **ancho** y **área**:

```
int ancho; // así se declara otra variable  
int área; // ésta es una tercera variable
```

Advierte que cada una de estas últimas utiliza el mismo formato que la primera, solamente cambia el nombre de la variable.

En general, para declarar una variable utilizarás una declaración como la siguiente:

*tipo nombre-de-variable*;

Aquí, *tipo* especifica el tipo de variable que será declarada, y *nombre-de-variable* es el nombre que deseas asignarle. Además de **int**, C# soporta otros tipos de datos como **double**, **char** y **string**.

La siguiente línea de código asigna a la variable **largo** el valor 9:

```
largo = 9;
```

En C# el operador de asignación es el signo de igual, que copia el valor de su lado derecho a la variable que se encuentra a su izquierda.

La siguiente línea de código envía como datos de salida el valor de la variable **largo** antecedida por la cadena de caracteres “el largo es de.”

```
Console.WriteLine("el largo es de " + largo);
```

En esta declaración, el signo de suma (+) indica que el valor de **largo** sea mostrado después de la cadena de caracteres que le antecede. Este enfoque puede ser generalizado: con el operador + puedes concatenar tantos elementos como deseas dentro de una misma declaración **WriteLine()**.

A continuación, se asigna 7 como valor de **ancho** utilizando el mismo tipo de declaración ya descrita. Después, la siguiente línea de código asigna a la variable **área** el valor de **largo** multiplicado por **ancho**.

```
área = ancho * largo;
```

Esta línea multiplica el valor que almacena la variable **largo** (que es 9) por el valor almacenado en la variable **ancho** (que es 7), para después almacenar el resultado en la variable **área**. De esta manera, después de ejecutar la línea, la variable **área** contiene el valor de 63. Los valores de **largo** y **ancho** permanecerán sin cambios.

Como la mayoría de los lenguajes de computación, C# soporta un conjunto completo de operadores aritméticos, incluyendo los que se muestran a continuación:

+	Suma
-	Resta
*	Multiplicación
/	División

Todos ellos pueden ser utilizados con cualquier tipo de dato numérico.

He aquí las dos siguientes líneas del programa:

```
Consol e. Write("el área de ancho * largo: ");  
Consol e. Wri teLi ne(área);
```

### Pregunta al experto

**P:** Ha dicho que todas las variables deben declararse antes de ser utilizadas, y que todas las variables tienen un tipo específico. Sin embargo, mencionó que C# 3.0 incluye una nueva característica llamada *variable de tipo implícito*. ¿Qué es eso? ¿Evita la necesidad de declarar variables?

**R:** Como aprenderás en el capítulo 2, las variables de tipo implícito son aquellas cuyo tipo es determinado automáticamente por el compilador. Debes entender, sin embargo, que este tipo de variables de cualquier manera deben declararse. En lugar de utilizar un nombre de tipo, como **int**, una variable de tipo implícito es declarada utilizando la palabra clave **var**. Las variables de tipo implícito son de gran utilidad en diferentes situaciones especializadas (en especial las que están relacionadas con LINQ), pero no están planeadas para sustituir las variables de tipo explícito en general. Normalmente, cuando declares una variable debes asignarle un tipo explícito.

Dos nuevos eventos ocurren aquí. Primero, el método integrado **Write()** se utiliza para mostrar la cadena de caracteres “el área de ancho \* largo:”. Esta cadena de caracteres *no* es seguida por una nueva línea. Esto significa que cuando se generen los próximos datos de salida, comenzarán en la misma línea. El método **Write()** es muy similar al método **WriteLine()**, sólo que el primero no genera una nueva línea después de cada invocación. Segundo, advierte que en la invocación a **WriteLine()** la variable área se utiliza directamente. Tanto **Write()** como **WriteLine()** pueden utilizarse para presentar valores de salida de cualquier tipo integrado en C#.

Un punto más sobre la declaración de variables antes de seguir adelante: es posible exponer dos o más variables utilizando la misma declaración. Simplemente separa sus nombres con comas. Por ejemplo: **largo** y **ancho** pudieron declararse de la siguiente manera:

```
int largo, ancho; // tanto largo como ancho se exponen utilizando la misma declaración
```

## El tipo de dato Doble

En el ejemplo anterior se utilizó una variable de tipo **int**. Sin embargo, este tipo de variables sólo puede contener números enteros, y por lo mismo no pueden ser utilizadas cuando se requiere un componente decimal. Por ejemplo, una variable **int** puede contener el valor 18, pero no el valor 18.3. Por fortuna, **int** es sólo uno de los muchos tipos de datos definidos en C#. Para permitir el uso de valores con componentes decimales, C# define dos tipos de valores flotantes: **float** y **double**, que representan valores de precisión sencilla y de doble precisión, respectivamente. De los dos, **double** es el de mayor uso.

Para declarar una variable de tipo **double**, utiliza la declaración similar a la que se muestra a continuación:

```
double resultado;
```

En el ejemplo, **resultado** es el nombre de la variable, la cual es de tipo **double**. Como **resultado** tiene un tipo de valor flotante, puede contener valores como 122.23, 0.034 y -19.0.

Para comprender mejor la diferencia entre **int** y **double** prueba el siguiente programa:

```
/*
Este programa muestra las diferencias
entre int y double
*/
using System;

class IntVsDouble {
    static void Main() {
        int ivar; // instrucción que declara una variable int
        double dvar; // instrucción que declara una variable double
        ivar = 10;
        dvar = 10.0;
        // Asignar a ivar el valor de 10.
        // Asignar a dvar el valor de 10.0.
    }
}
```

*ivar* es de tipo **int**.  
*dvar* es de tipo **double**.

## 26 Fundamentos de C# 3.0

```
Console.WriteLine("Valor original de ivar: " + ivar);
Console.WriteLine("Valor original de dvar: " + dvar);

Console.WriteLine(); // inserta una línea en blanco ← Envía a la
                     consola una línea en blanco.

// Ahora divide los dos entre 4.
ivar = ivar / 4; ← Esta es una división sin residuo.
dvar = dvar / 4.0; ← Esta división conserva el componente decimal.

Console.WriteLine("ivar después de la división: " + ivar);
Console.WriteLine("dvar después de la división: " + dvar);
}
```

Los datos de salida de este programa son los siguientes:

```
Valor original de ivar: 10
Valor original de dvar: 10

ivar después de la división: 2
dvar después de la división: 2.5
```

Como puedes ver, cuando **ivar** es dividida entre 4, se realiza una división entre números enteros sin residuo y el resultado es 2; el componente decimal se pierde. Sin embargo, cuando **dvar** se divide entre 4.0, el componente decimal se conserva y el resultado es 2.5.

Como lo muestra el programa, cuando quieras especificar un valor de punto flotante en un programa, debes incluir el punto decimal. Si no lo haces será interpretado como un entero. Por ejemplo, en C# el valor 100 es un entero y 100.0 es un valor de punto flotante.

En el programa anterior hay un elemento nuevo que debes tomar en cuenta: para insertar una línea en blanco, simplemente invoca **WriteLine()** sin argumentos.

### Pregunta al experto

**P:** ¿Por qué tiene C# diferentes tipos de datos para enteros y valores de punto flotante? Es decir, ¿por qué no son del mismo tipo todos los valores numéricos?

**R:** C# proporciona diferentes tipos de datos para que escribas programas eficientes. Por ejemplo, realizar operaciones aritméticas con valores enteros es mucho más rápido que los cálculos realizados con valores de punto flotante. De esta manera, si no requieres manejar fracciones, no necesitas recurrir a los complejos cálculos que implica el uso de los tipos **float** y **double**. En segundo lugar, la cantidad de memoria requerida por un tipo de datos puede ser menor a la requerida por otro. Ofreciendo diferentes tipos de datos, C# te permite optimizar los recursos de la computadora. Finalmente, algunos algoritmos requieren el uso específico de un tipo de dato (o al menos se beneficia de él). C# ofrece varios tipos de datos integrados para darte la mayor flexibilidad.

**Prueba esto**

## Convertir Fahrenheit a Celsius

Aunque el programa de ejemplo anterior muestra muchas importantes características del lenguaje C#, no son de gran utilidad. Si bien es cierto que en este punto tus conocimientos sobre C# son precarios, aún así puedes utilizar los conocimientos que has adquirido para crear un programa de utilidad práctica. A continuación crearás un programa que convierte los grados Fahrenheit en grados Celsius.

El programa declara dos variables tipo **double**. Una de ellas contendrá los grados Fahrenheit y la otra contendrá el equivalente en grados Celsius después de la conversión. La fórmula para convertir Fahrenheit a Celsius es la siguiente:

$$C = \frac{5}{9} * (F - 32)$$

donde C es la temperatura expresada en grados Celsius y F la representada en grados Fahrenheit.

### Paso a paso

1. Crea un nuevo archivo C# y nómbralos **FtoC.cs**. Si estás utilizando el IDE de Visual Studio en lugar del compilador de línea de comandos, necesitarás añadir este archivo a un proyecto C#, como se describió anteriormente.<sup>1</sup>
2. Inserta el siguiente programa al archivo creado:

```
/*
Este programa convierte grados Fahrenheit a Celsius.
Nombre este programa FtoC.cs.
*/
using System;

class FtoC {
    static void Main() {
        double f; // Contiene la temperatura en grados Fahrenheit
        double c; // Contiene la temperatura en grados Celsius

        // Comienza con 59 grados Fahrenheit
        f = 59.0;

        // convierte a Celsius
        c = 5.0 / 9.0 * (f - 32.0);

        Console.WriteLine(f + " grados Fahrenheit son ");
        Console.WriteLine(c + " grados Celsius.");
    }
}
```

<sup>1</sup> Si usas el IDE, es indispensable que crees un nuevo proyecto; si añades el archivo ejemplo a un proyecto existente, el compilador marcará un error porque tendrás dos constructores Main().

- 3.** Compila el programa utilizando el IDE de Visual Studio (de acuerdo con las instrucciones mostradas anteriormente en este mismo capítulo) o bien utilizando la siguiente línea de comandos:

```
C>csc FtoC.cs
```

- 4.** Ejecuta el programa y verás los siguientes datos de salida:

```
59 grados Fahrenheit son 15 grados Celsius.
```

- 5.** Como es su objetivo, este programa convierte 59 grados Fahrenheit en su equivalente a Celsius, pero puedes cambiar el valor de **f** para convertir cualquier temperatura. Advierte que las variables **f** y **c** son de tipo **double**. Esto es necesario porque se requiere el componente decimal para asegurar una conversión precisa.
- 

## Dos declaraciones de control

Dentro de un método, las declaraciones se ejecutan una detrás de otra, de arriba hacia abajo. Sin embargo, es posible alterar este orden con el uso de varias declaraciones de control del programa soportadas por C#. Aunque veremos con detalle las declaraciones de control en el capítulo 3, dos de ellas son introducidas brevemente en este punto porque las utilizaremos para escribir programas de ejemplo.

### La declaración if

Puedes seleccionar una parte del programa para ser ejecutada a través del uso de la declaración condicional **if** de C#. La declaración **if** en C# funciona de manera muy parecida a su equivalente IF de cualquier otro lenguaje de programación. Por ejemplo, su sintaxis es similar a la declaración **if** de C, C++ y Java. Su forma más sencilla es la siguiente:

```
if(condición) declaración;
```

Aquí, *condición* es una expresión booleana (es decir, verdadero o falso). Si la *condición* es verdadera se ejecuta la *declaración*. Si la *condición* es falsa la *declaración* se omite. He aquí un ejemplo:

```
if(10 < 11) Consol e.WriteLine("10 es menor que 11");
```

En este caso, como 10 es menor que 11, la expresión condicional es verdadera y se ejecutará el método **WriteLine()**. Ahora analiza el siguiente ejemplo:

```
if(10 < 9) Consol e.WriteLine("este mensaje no se mostrará");
```

En tal caso, 10 no es mayor que 9 y por tanto el método **WriteLine()** no se ejecutará.

C# define un complemento completo de operadores relacionales que pueden utilizarse en las expresiones condicionales. La siguiente tabla los muestra:

Operador	Significado
<	Menor que
<=	Menor que o igual a
>	Mayor que
>=	Mayor que o igual a
==	Igual a
!=	Diferente de

Advierte que el operador relacional de igualdad está conformado por un doble signo de igual.

A continuación tenemos un programa que ilustra la declaración **if** y los operadores relacionales.

// Muestra la función de if.

```
using System;
class ifDemo {
    static void Main() {
        int a, b, c;
        a = 2;
        b = 3;
        // Esta declaración es exitosa porque a es menor que b.
        if (a < b) Console.WriteLine("a es menor que b");
        // Esto no mostrará nada porque a no es igual a b.
        if(a == b) Console.WriteLine("no verás esto");
        // Esta declaración es exitosa porque el valor de a es 2.
        if(a == 2) Console.WriteLine("el valor de a es igual a 2");
        // Esta declaración fallará porque el valor de a no es igual a 19.
        if(a == 19) Console.WriteLine("no verás esto");
        // Esta declaración será exitosa porque a es igual a b - 1.
        if(a == b-1) Console.WriteLine("a es igual a b - 1");
        Console.WriteLine();
        c = a - b; // el valor de c es -1
        Console.WriteLine("el valor de c es -1");
        if(c >= 0) Console.WriteLine("c es un número positivo");
        if (c < 0) Console.WriteLine("c es un número negativo");
        Console.WriteLine();
        c = b - a; // el valor de c es igual a 1
    }
}
```

La declaración **WriteLine()** se ejecuta sólo cuando la condición es verdadera.



↑ Ésta es la condición que prueba **if**.

```
Console.WriteLine("el valor actual de c es igual a 1");
if(c >= 0) Console.WriteLine("c es un número positivo");
if(c < 0) Console.WriteLine("c es un número negativo");

Console.WriteLine();
}
```

Los datos de salida generados por el programa son los siguientes:

```
a es menor que b
el valor de a es igual a 2
a es igual a b - 1

el valor de c es -1
c es un número negativo

el valor actual de c es igual a 1
c es un número positivo
```

Debes advertir otro aspecto de este programa. La línea

```
int a, b, c;
```

declara tres variables: **a**, **b** y **c**, utilizando comas para separar cada una. Como se mencionó anteriormente, cuando necesitas dos o más variables del mismo tipo todas ellas pueden ser expresadas en la misma declaración, simplemente separa los nombres con comas.

## El loop **for**

Puedes ejecutar repetidas veces una secuencia de código creando una instrucción de reiteración llamada *loop*. C# proporciona un gran surtido de constructores para loop. El que veremos aquí será el loop **for**. Si estás familiarizado con C, C++ o Java, te complacerá saber que el loop **for** en C# funciona de la misma manera como lo hace en aquellos lenguajes. La forma más simple del loop **for** se muestra a continuación:

```
for(inicialización; condición; reiteración) declaración;
```

En su forma más común, la parte de *inicialización* del loop configura el control de la variable en su valor inicial. La *condición* es una expresión booleana que prueba el control de la variable. Si el resultado de esa prueba es verdadero, el loop **for** continúa sus reiteraciones; en caso de ser falso, el loop termina. La expresión *reiteración* determina cómo cambiará la variable de control en cada reiteración. Aquí tenemos un pequeño programa que muestra el funcionamiento del loop **for**:

```
//Demostración del loop for.

using System;
class ForDemo {
    static void Main() {
        int cuenta;
```

```

Console.WriteLine("Contar del 0 al 4:");
for (cuenta = 0; cuenta < 5; cuenta = cuenta+1)
    Console.WriteLine("La cuenta es igual a " + cuenta);
Console.WriteLine("Fin.");
}

```

Si **cuenta** es menor que 5, se ejecuta la declaración **WriteLine( )**.

Inicializa el valor de **cuenta** en cero.

Suma 1 al valor actual de **cuenta** cada vez que se ejecuta una reiteración.

Los datos de salida generados por el programa son los siguientes:

```

Contar del 0 al 4
la cuenta es igual a 0
la cuenta es igual a 1
la cuenta es igual a 2
la cuenta es igual a 3
la cuenta es igual a 4
Fin.

```

En este ejemplo, **cuenta** es la variable de control del loop. En la inicialización del loop el valor de esta variable **for** es igual a cero. Al principio de cada reiteración (incluyendo la primera) se aplica la prueba condicional **cuenta < 5**. Si el valor resultante es verdadero, se ejecuta la declaración **WriteLine( )**, y a continuación se ejecuta la porción de reiteración del loop. El proceso se repite hasta que la prueba condicional arroje un valor falso; en ese punto se ejecuta la parte final del loop.

Como información de interés, diremos que en los programas C# escritos profesionalmente casi nunca encontrarás la porción de reiteración escrita como en el ejemplo anterior. Es decir, muy rara vez encontrarás una declaración como ésta:

```
cuenta = cuenta + 1;
```

La razón es que C# incluye un operador de incremento especial que realiza la misma operación de manera más compacta. El operador de incremento es **++** (dos signos de suma consecutivos). El operador de incremento suma uno al valor actual de su operando. Utilizando este operador, el ejemplo anterior debería escribirse así:

```
cuenta++;
```

De tal manera que el loop **for** del programa anterior por lo general debe escribirse así:

```
for (cuenta = 0; cuenta < 5; cuenta++)
```

Puedes intentar hacer el cambio. Como verás, el loop se ejecuta exactamente de la misma manera como lo hacía antes.

C# también proporciona un operador de decremento, que se escribe **--** (dos signos de resta consecutivos). Este operador resta uno al valor actual de su operando.

## Utilizar bloques de código

Otro elemento clave de C# es el *bloque de código*, que es un conjunto de declaraciones agrupadas. Un bloque de código se crea encerrando las declaraciones entre una llave de apertura y una de clausura. Una vez que se ha creado un bloque de código, se convierte en una unidad lógica que

puede utilizarse de la misma manera en que se utiliza una declaración sencilla. Por ejemplo, un bloque puede ser utilizado con declaraciones **if** y **for**. Analiza esta declaración **if**:

```
if (contador < max) {  
    userCount = contador;  
    del aytme = 0;  
}
```

En este caso, si **contador** es menor que **max**, se ejecutarán las dos declaraciones que conforman el bloque. Así, las dos declaraciones que se encuentran dentro de las llaves forman una sola unidad lógica, y una de ellas no puede ejecutarse si no se ejecuta la otra. El punto clave aquí es que cuando necesites vincular lógicamente dos o más declaraciones, lo haces creando un bloque. Los bloques de código permiten la implementación de muchos algoritmos con gran claridad y eficiencia.

A continuación presentamos un programa que utiliza bloques de código para evitar una división entre cero:

```
// Muestra un bloque de código.  
  
using System;  
class BloqueDemo {  
    static void Main() {  
        double i, j, d;  
        i = 5.0;  
        j = 10.0;  
  
        // Este if se aplica a un bloque.  
        if(i != 0) {  
            Console.WriteLine("i no es igual a cero");  
            d = j / i;  
            Console.WriteLine("j / i es " + d);  
        }  
    }  
}
```

← El **if** se aplica a todo el bloque.

Los datos de salida generados por este programa son:

```
i no es igual a cero  
j / i es 2
```

En este caso, el **if** se aplica a todo el bloque y no a una declaración individual. Si la condición que controla el **if** es verdadera (y en este caso lo es), las tres declaraciones que conforman el bloque serán ejecutadas. Cambia el valor de **i** a cero y observa el resultado.

Como verás más adelante en este libro, los bloques de código tienen propiedades y usos adicionales. Sin embargo, la principal razón de su existencia es crear unidades de código lógicamente inseparables.

## Punto y coma, y posiciones

En C# el punto y coma (;) señala el final de una declaración. Esto quiere decir que cada declaración individual debe concluir con un punto y coma.

## Pregunta al experto

**P:** ¿El uso de bloques de código puede producir alguna ineficiencia durante el tiempo de ejecución? En otras palabras, ¿la llave de apertura ({}) y la de clausura (}) consumen tiempo extra durante la ejecución del programa?

**R:** No. Los bloques de código no sobrecargan el programa de ninguna manera. De hecho, dada su capacidad de simplificar el código de ciertos algoritmos, su uso suele incrementar la velocidad y eficiencia del programa.

Como sabes, un bloque es un conjunto de declaraciones conectadas lógicamente y encerradas entre una llave de apertura ({}) y una de clausura (}). Un bloque *no* concluye con un punto y coma. Como un bloque es un grupo de declaraciones, tiene sentido que no termine con un punto y coma; en lugar de ello el final del bloque se indica con la llave de clausura.

C# no reconoce el final de la línea como el final de la declaración; solamente un punto y coma es reconocido como su conclusión. Por esta razón, la cantidad de líneas no es necesariamente proporcional a la cantidad de declaraciones. Por ejemplo, para C# la declaración

```
x = y;  
y = y + 1;  
Consol e. Wri teLi ne(x + " " + y);
```

es lo mismo que

```
x = y; y = y + 1; Consol e. WriteLi ne(x + " " + y);
```

Más aún, los elementos individuales de una declaración también pueden escribirse en líneas separadas. Por ejemplo, la siguiente es completamente aceptable:

```
Consol e. Writ el i ne("Ésta es una lí nea larga con datos de sal i da" +  
                     x + y + z +  
                     "más i nf ormac i ón");
```

Dividir líneas de esta manera se utiliza a menudo para hacer más legibles los programas. También ayuda a evitar que las líneas demasiado largas sean cortadas automáticamente por el procesador de textos que se utiliza.

## Prácticas de indentado

Habrás notado en los ejemplos previos que ciertas declaraciones se separan sensiblemente del margen izquierdo y conservan esa alineación en los renglones posteriores; a eso se le llama *indentación*. C# es un lenguaje de forma libre; esto significa que no importa dónde coloques unas declaraciones respecto a otras en una línea. Sin embargo, a lo largo de los años se ha desarrollado un

estilo de indentación común y aceptable que permite escribir los programas de manera muy clara. Este libro sigue ese estilo y es recomendable que tú también lo hagas. Para hacerlo, debes aplicar la indentación un nivel después de cada llave de apertura y regresar al nivel anterior después de la llave de clausura. Existen algunas declaraciones que merecen una indentación adicional; las veremos más adelante.

**Prueba esto**

## Mejorar el programa de conversión de temperatura

Puedes utilizar el loop **for**, la declaración **if** y los bloques de código para crear una versión mejorada de tu convertidor de grados Fahrenheit a Celsius que desarrollaste en el primero de estos ejercicios. Esta nueva versión presentará una tabla de conversiones, comenzando con 0 grados Fahrenheit y concluyendo con 99. Después de cada 10 grados se insertará una línea en blanco. Esto se consigue con el uso de una variable llamada **contador** que, como su nombre lo indica, cuenta el número de líneas que han sido enviadas como datos de salida. Presta especial atención a su uso.

### Paso a paso

1. Crea un nuevo archivo llamado **FaCTabla.cs**.
2. Copia el siguiente código en el archivo:

```
/*
Este programa muestra una tabla de conversión
de grados Fahrenheit a Celcius.

Nombra este programa FtoCTabla.cs.
*/
using System;

class FtoCTabla {
    static void Main() {
        double f, c;
        int contador;

        contador = 0;
        for(f = 0.0; f < 100.0; f++) {
            // Convertir a Celcius
            c = 5.0 / 9.0 * (f - 32.0);

            Console.WriteLine(f + " grados Fahrenheit son " +
                c + " grados Celcius.");

            contador++;
            // Cada 10 líneas inserta una línea en blanco.
            if(contador == 10) {
```

```
Console.WriteLine();
    contador = 0; // Reinicia el contador de líneas
}
}
}
```

**3.** Compila este programa como se describió anteriormente.

**4.** Ejecuta el programa. A continuación hallarás una porción de los datos de salida que verás. Advierte que cada décima línea se inserta una en blanco. Como se mencionó, esta tarea es controlada por la variable **contador**, que inicia en cero. Cada vez que se ejecuta el loop **for**, el **contador** se incrementa. Cuando el **contador** llega a 10 se inserta una línea en blanco y el **contador** se reinicia desde cero. Este proceso hace que los datos de salida queden ordenados en grupos de diez unidades.

```
0 grados Fahrenheit son -17. 7777777777778 grados Celsius.
1 grados Fahrenheit son -17. 2222222222222 grados Celsius.
2 grados Fahrenheit son -16. 6666666666667 grados Celsius.
3 grados Fahrenheit son -16. 1111111111111 grados Celsius.
4 grados Fahrenheit son -15. 5555555555556 grados Celsius.
5 grados Fahrenheit son -15 grados Celsius.
6 grados Fahrenheit son -14. 4444444444444 grados Celsius.
7 grados Fahrenheit son -13. 8888888888889 grados Celsius.
8 grados Fahrenheit son -13. 3333333333333 grados Celsius.
9 grados Fahrenheit son -12. 7777777777778 grados Celsius.

10 grados Fahrenheit son -12. 2222222222222 grados Celsius.
11 grados Fahrenheit son -11. 6666666666667 grados Celsius.
12 grados Fahrenheit son -11. 1111111111111 grados Celsius.
13 grados Fahrenheit son -10. 5555555555556 grados Celsius.
14 grados Fahrenheit son -10 grados Celsius.
15 grados Fahrenheit son -9. 4444444444444 grados Celsius.
16 grados Fahrenheit son -8. 8888888888888 grados Celsius.
17 grados Fahrenheit son -8. 3333333333333 grados Celsius.
18 grados Fahrenheit son -7. 7777777777778 grados Celsius.
19 grados Fahrenheit son -7. 2222222222222 grados Celsius.
```

---

## Las palabras clave de C#

En sus cimientos, un lenguaje de computación es definido por sus palabras clave, y C# ofrece un conjunto rico y diverso. C# define dos tipos generales de palabras clave: *reservadas* y *contextuales*. Las reservadas no pueden ser utilizadas como nombres de variable, clases ni métodos; sólo pueden ser utilizadas como palabras clave, por eso reciben el nombre de *reservadas*. En ocasiones también reciben el nombre de *términos reservados* e *identificadores reservados*. Actualmente existen 77 palabras clave reservadas definidas en la versión 3.0 del lenguaje C# 3.0. Todas ellas se muestran en la tabla 1-1.

C# 3.0 define 13 palabras clave contextuales que tienen un significado especial en ciertos contextos. En tales contextos, actúan como palabras clave y fuera de ellos pueden ser utilizadas como nombres para otros elementos del programa, como las variables. Así pues, técnicamente no son palabras reservadas, pero como regla general debes considerar las palabras clave contextuales como reservadas y evitar su uso para cualquier otro propósito diferente al que le asigna el lenguaje. Utilizar palabras clave contextuales como nombre para cualquier otro elemento del programa puede causar confusión y es considerada una mala práctica por la mayoría de los programadores. Las palabras clave contextuales aparecen en la tabla 1-2.

## Identificadores

En C# un identificador es un nombre asignado a un método, a una variable o a cualquier otro elemento definido por el usuario. Los identificadores pueden tener una longitud variada. Los nombres de las variables pueden comenzar con cualquier letra del alfabeto o con un guión bajo; después pueden continuar con más letras, números o guiones bajos. Los guiones bajos pueden utilizarse para incrementar la legibilidad de un nombre de variable, como **cuenta\_línea**.

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

**Tabla 1-1** Las palabras clave reservadas de C#

Las mayúsculas y las minúsculas son diferentes, lo cual quiere decir que para C# **mivar** y **MiVar** son dos nombres diferentes. He aquí algunos ejemplos de identificadores aceptables:

Test	x	y2	MaxCarga
arriba	_tope	mi_var	ejemplo23

Recuerda que no puedes iniciar un identificador con un número, por lo que **12x** es inválido, por ejemplo. Las buenas prácticas de programación indican que utilices nombres de indicadores que reflejen el significado o el uso del elemento al que pertenecen.

Aunque no puedes utilizar ninguna palabra clave de C# como identificador, el lenguaje te permite colocar una arroba (@) antes de la palabra clave para convertirla en un identificador legal. Por ejemplo, **@for** es un identificador legal; en este caso, el identificador es de hecho la palabra **for** y la @ es ignorada. Sinceramente, el uso de la arroba para legalizar palabras clave no es recomendable, salvo propósitos muy especiales. La arroba también puede anteceder a cualquier identificador, pero se considera una mala práctica.

from	get	group	into	join
let	orderby	partial	select	set
value	where	yield		

**Tabla 1-2** Las palabras clave contextuales de C#

## La biblioteca de clases de C#

Los programas ejemplo mostrados en este capítulo utilizan dos métodos integrados de C#: **WriteLine()** y **Write()**. Como mencionamos, estos métodos son miembros de la clase **Console**, la cual es parte de la nomenclatura **System**, que a su vez está definido en la biblioteca de clases .NET Framework. Como se explicó anteriormente en este capítulo, el ambiente de desarrollo de C# depende de la biblioteca de clases de .NET Framework para proporcionar soporte a tareas como datos de entrada y salida (I/O), manejo de cadenas de caracteres, trabajo en red e interfaces gráficas de usuario (GUI por sus siglas en inglés). De esta manera, como totalidad C# es una combinación del lenguaje en sí en combinación con las clases estándar de .NET. Como verás, la biblioteca de clases proporciona mucha de la funcionalidad que es parte de cualquier programa C#. En efecto, parte de la tarea de convertirse en un programador de C# es aprender a usar la biblioteca estándar. A lo largo de este libro describimos varios elementos de clases y métodos pertenecientes a la biblioteca de clases .NET. Sin embargo, la biblioteca .NET es muy extensa, y es algo que querrás explorar más a fondo por cuenta propia.



## Autoexamen Capítulo 1

- 1.** ¿Qué es MSIL y por qué es importante para C#?
- 2.** ¿Qué es el lenguaje común de tiempo de ejecución?
- 3.** ¿Cuáles son los tres principios fundamentales de la programación orientada a objetos?
- 4.** ¿Dónde comienzan su ejecución los programas C#?
- 5.** ¿Qué es una variable? ¿Qué es una nomenclatura?
- 6.** ¿Cuáles de los siguientes nombres de variable son inválidos?
  - A.** cuenta
  - B.** \$cuenta
  - C.** cuenta27
  - D.** 67cuenta
  - E.** @if
- 7.** ¿Cómo se crea un comentario de una sola línea? ¿Cómo se crea un comentario de varias líneas?
- 8.** Escribe la forma general de una declaración **if**. Escribe la forma general de un loop **for**.
- 9.** ¿Cómo creas un bloque de código?
- 10.** ¿Es necesario comenzar cada programa C# con la siguiente declaración?  
using System;
- 11.** La gravedad en la Luna es aproximadamente un 17 por ciento de la gravedad en la Tierra. Escribe un programa que calcule tu peso en la Luna.
- 12.** Adapta el programa FaCTable.cs que aparece en “Prueba esto: mejorar el programa de conversión de temperatura”, de manera que presente una tabla de conversión de pulgadas a metros. Muestra 12 pies de conversión, pulgada por pulgada. Inserta una línea en blanco cada 12 pulgadas. (Un metro equivale aproximadamente a 39.37 pulgadas.)

# Capítulo 2

Introducción a los tipos  
de datos y operadores

## Habilidades y conceptos clave

- Tipos básicos de C#
  - Formato de datos de salida
  - Literales
  - Inicializar variables
  - Reglas de alcance de un método
  - Conversión y transformación de tipos
  - Operadores aritméticos
  - Operadores lógicos y relacionales
  - El operador de asignación
  - Expresiones
- 

**L**os cimientos de todo lenguaje de programación están conformados por los tipos de datos y los operadores, y C# no es la excepción. Estos elementos definen los límites de un lenguaje y determinan los tipos de tareas a las que se pueden aplicar. Como podrías esperar, C# soporta una rica variedad de ambos, tanto tipos de datos como operadores, haciéndolo apropiado para la creación de un amplio rango de programas.

Los tipos de datos y los operadores son un tema muy amplio. Comenzaremos aquí con una explicación de los tipos de datos fundamentales de C# y los operadores de mayor uso. También analizaremos muy de cerca las variables y las expresiones.

## Por qué son importantes los tipos de datos

Los tipos de datos son particularmente importantes en C# porque se trata de un lenguaje fuertemente tipificado. Esto significa que en todas las operaciones el tipo de datos es verificado por el compilador para comprobar su compatibilidad con la operación que quiere realizarse. Los operadores ilegales no serán compilados. De esta manera, la verificación estricta de tipos ayuda a prevenir errores y enriquece la confiabilidad del programa. Para poder realizar la verificación estricta de los elementos, todas las variables, expresiones y valores deben corresponder a un determinado tipo; no existe, por ejemplo, el concepto de una variable “sin tipo”. Más aún, el tipo de un valor determina las operaciones que le está permitido realizar, por lo que las operaciones que están permitidas a cierto tipo de datos pueden estar prohibidas para otros.

## Tipos de valor C#

C# cuenta con dos categorías generales de tipos de datos integrados: *tipos de valor* y *tipos de referencia*. La diferencia entre uno y otro es el contenido de la variable. En un tipo de valor la variable contiene, de hecho, un valor, como 101 o 98.6. En un tipo de referencia, la variable contiene una

referencia al valor. El tipo de referencia más común es la clase, pero por el momento aplazaremos el análisis de las clases y los tipos de referencia para concentrarnos en los tipos de valor.

En la parte medular de C# existen los 13 tipos de valor que se muestran en la tabla 2-1. En conjunto, reciben el nombre de *tipos simples*. Y se les llama así porque consisten en un solo valor (en otras palabras, no son el resultado de la combinación de dos o más valores). Estos tipos conforman los fundamentos del sistema de tipos de C#, proporcionando los elementos básicos, de bajo nivel, sobre los cuales opera el programa. Los tipos simples también son conocidos como *tipos primitivos*.

C# especifica estrictamente un rango y un comportamiento a cada uno de estos tipos simples. Dados los requerimientos de portabilidad y soporte para programación de lenguajes cruzados, C# es inflexible en esta materia. Por ejemplo, un **int** es el mismo en todos los ambientes de ejecución, sin necesidad de reescribir el código para acoplarse a una plataforma específica. Aunque la especificación estricta del tamaño de los tipos simples puede causar una pequeña disminución en el rendimiento del programa en algunos ambientes, es necesaria para alcanzar los requerimientos de portabilidad.

### NOTA

Además de los tipos simples, C# define otras tres categorías de tipos de valor. Tales categorías son: enumeraciones, estructuras y tipos anulables, todas ellas abordadas más tarde en este libro.

## Enteros

C# define nueve tipos de enteros: **char**, **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long** y **ulong**. Sin embargo, el tipo **char** es utilizado principalmente para representar caracteres y es analizado más

Tipo	Significado
<b>bool</b>	Representa valores verdadero/falso
<b>byte</b>	Entero sin negativos de 8 bits
<b>char</b>	Carácter
<b>decimal</b>	Tipo numérico para cálculos financieros
<b>double</b>	Punto flotante de doble precisión
<b>float</b>	Punto flotante de precisión sencilla
<b>int</b>	Entero
<b>long</b>	Entero largo
<b>sbyte</b>	Entero con negativos de 8 bits
<b>short</b>	Entero corto
<b>uint</b>	Entero sin negativos
<b>ulong</b>	Entero largo sin negativos
<b>ushort</b>	Entero corto sin negativos

**Tabla 2-1** Los tipos simples de C#

tarde en este capítulo. Los restantes ocho tipos de enteros se utilizan para cálculos numéricos. Su amplitud en bits y rango se muestran en la siguiente tabla:

Tipo	Amplitud en bits	Rango
byte	8	0 a 255
sbyte	8	-128 a 127
short	16	-32,768 a 32,767
ushort	16	0 a 65,535
int	32	-2,147,483,648 a 2,147,483,647
uint	32	0 a 4,294,967,295
long	64	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
ulong	64	0 a 18,446,744,073,709,551,615

Como lo muestra la tabla, C# define versiones con y sin negativos de varios tipos de entero. La diferencia entre enteros con y sin negativos la determina el bit de alto orden al momento de su interpretación. Si se especifica un entero con negativos, el compilador de C# generará un código que da por hecho que se debe utilizar un bit de alto orden como identificador de signo negativo o positivo para el entero en cuestión. Si el valor del bit identificador es 0, el número es positivo; en caso de que el bit identificador sea 1, entonces el entero es negativo. Casi siempre los números negativos son representados utilizando el método llamado *complemento doble*. En este método, se invierten todos los bits del número para añadir luego el bit de alto orden con valor 1.

Los enteros con negativos son importantes para la gran mayoría de los algoritmos, pero sólo tienen la mitad de la magnitud absoluta de sus contrapartes sin negativos. Por ejemplo, un tipo **short** aquí es 32,767:

01111111 11111111

Para un valor con negativos, si el bit de alto orden fuera establecido en 1, entonces el número sería interpretado como -1 (considerando el formato complemento doble). Sin embargo, si lo declararas como **ushort**, entonces cuando el bit de alto orden fuera establecido como 1 el número se convertiría en 65,535.

Probablemente el tipo de entero más utilizado sea **int**. Las variables de tipo **int** son utilizadas generalmente para controlar reiteraciones (loop), para indexar arreglos y para operaciones matemáticas de propósito general donde se ocupan números naturales. Cuando necesites un entero con un rango superior al de **int** tienes diversas opciones. Si el valor que quieras almacenar no necesita negativos, puedes utilizar **uint**. Para valores que requieren negativos superiores al rango de **int** puedes usar **long**. Para valores más grandes que no requieran negativos, **ulong** es la opción.

A continuación presentamos un programa que calcula la cantidad de pulgadas cúbicas de un cubo de una milla por lado. Como se trata de un número muy grande, el programa utiliza una variable **long** para almacenarlo.

```
// Calcula la cantidad de pulgadas cúbicas en una milla cúbica.

using System;
```

```
class Pulgadas {
    static void Main() {
        long ci;
        long im; ← Declara dos variables long.
        im = 5280 * 12;
        ci = im * im * im;
        Console.WriteLine("Hay " + ci +
                           " pulgadas cúbicas en una milla cúbica.");
    }
}
```

La respuesta del programa es la siguiente:

```
Hay 254358061056000 pulgadas cúbicas en una milla cúbica.
```

Claramente el resultado no podía almacenarse en una variable **int**, ni en una **uint**.

Los tipos de entero más pequeños son **byte** y **sbyte**. El tipo **byte** es un valor sin negativos entre 0 y 255. Las variables tipo **byte** son especialmente útiles cuando se trabaja con datos binarios sin procesar, como un flujo de bytes producido por algún dispositivo. Para enteros pequeños con negativos utiliza **sbyte**. A continuación presentamos un ejemplo que utiliza una variable tipo **byte** para controlar un loop **for** que produce una sumatoria del número 100:

```
// Uso de byte.

using System;

class Usar_byte {
    static void Main() {
        byte x;
        int sum;

        sum = 0;
        for (x = 1; x <= 100; x++) ← Usa una variable byte
            sum = sum + x; ← para controlar el loop.

        Console.WriteLine("La sumatoria de 100 es " + sum);
    }
}
```

Los datos de salida que presenta el programa son:

```
La sumatoria de 100 es 5050
```

Como el loop **for** se ejecuta sólo de 0 a 100, se ajusta bien al rango de **byte** y no es necesario utilizar un tipo de variable más grande para controlarlo. Por supuesto, **byte** no puede ser utilizado para contener el resultado de la sumatoria porque 5050 rebasa por mucho su rango. Por eso la variable **sum** es un **int**.

Cuando necesites un entero que sea más grande que **byte** o **sbyte**, pero más pequeño que **int** o **uint**, utiliza **short** o **ushort**.

## Tipos de punto flotante

Como se explicó en el capítulo 1, los tipos de punto flotante pueden representar números con componentes decimales. Existen dos clases de tipos de punto flotante: **float** y **double**, que representan números de precisión simple y doble, respectivamente. El tipo **float** tiene una amplitud de 32 bits y un rango de 1.5E-45 a 3.4E+38. El tipo **double** tiene una amplitud de 64 bits y un rango de 5E-324 a 1.7E+308.

**Double** es el más usado de ambos. Una de las razones es porque muchas de las funciones matemáticas en la biblioteca de clases de C# (que es la biblioteca .NET Framework) utilizan valores **double**. Por ejemplo, el método **Sqrt()** (definido por la clase **System.Math**) regresa un valor **double** que es la raíz cuadrada de su argumento **double**. En el siguiente ejemplo **Sqrt()** es utilizado para calcular la longitud de una hipotenusa dadas las longitudes de los catetos.

```
/*
    Utiliza el teorema de Pitágoras para encontrar la longitud de la
    hipotenusa dadas las longitudes de los catetos.
*/
using System;

class Hipotenusa {
    static void Main() {
        double x, y, z;

        x = 3;
        y = 4;
        z = Math.Sqrt(x*x + y*y); ←———— Advierte cómo se invoca Sqrt(). El nombre
                                     del método es precedido por el nombre de
                                     la clase de la cual es miembro.
        Console.WriteLine("La hipotenusa es " + z);
    }
}
```

Los datos de salida que presenta el programa son los siguientes:

La hipotenusa es 5

Debemos notar otro punto del ejemplo anterior. Como se mencionó, **Sqrt()** es miembro de la clase **Math**. Advierte cómo es invocado **Sqrt()**: le antecede el nombre **Math**. Esto es similar a la forma como **Console** precede a **WriteLine()**. Aunque no todos los métodos son invocados especificando primero el nombre de la clase a la que pertenecen, en muchos casos sí es necesario.

## El tipo decimal

Quizás el tipo numérico más interesante de C# es **decimal**, el cual fue creado para utilizarse en cálculos monetarios. El tipo **decimal** utiliza 128 bits para representar valores en un rango de 1E-28 a 7.9E+28. Como probablemente lo sabes, la aritmética normal de punto flotante presenta

una serie de errores de redondeo cuando se aplica a valores decimales. El tipo **decimal** elimina esos errores y puede representar hasta 28 posiciones decimales (29 en algunos casos). Esta capacidad para representar valores decimales sin errores de redondeo lo hace especialmente útil para los cálculos que implican dinero.

He aquí un programa que utiliza el tipo **decimal** en un cálculo financiero. El programa calcula un balance antes de aplicar intereses.

```
// Uso del tipo decimal en un cálculo financiero.

using System;

class UsoDecimal {
    static void Main() {
        decimal balance;
        decimal tasa;

        // Calcular nuevo balance.
        balance = 1000.10m;           ← los valores decimal deben ir
        tasa = 0.1m;                 seguidos de una m o M.
        balance = balance * tasa + balance;

        Console.WriteLine("Nuevo balance: $" + balance);
    }
}
```

Los datos de salida presentados por este programa son:

```
Nuevo balance: $1100.110
```

Advierte que en este programa las constantes decimales van seguidas por el sufijo **m** o **M**. Esto es necesario porque sin ese sufijo los valores serían interpretados como constantes estándar de punto flotante, que no son compatibles con el tipo de dato **decimal**. Veremos con más detalle cómo especificar constantes numéricas más adelante en este mismo capítulo.

## Pregunta al experto

**P:** Los otros lenguajes de computación con los que he trabajado no tienen el tipo de dato **decimal**. ¿Es exclusivo de C#?

**R:** El tipo **decimal** no es permitido por C, C++ ni Java como tipo integrado. Por ello, dentro de la línea de descendencia directa, sí es un tipo único de C#.

## Caracteres

En C# los caracteres no son de 8 bits como en muchos otros lenguajes de computación, como C++. En lugar de ello, C# utiliza *Unicode*: el conjunto de caracteres que puede representar todas las grafías de todos los lenguajes humanos. Por ello, en C#, **char** es un tipo sin negativos de 16 bits con un rango de 0 a 65,535. El conjunto de caracteres ASCII estándar de 8 bits es un subconjunto de Unicode y su rango es de 0 a 127; por ello, los caracteres ASCII siguen siendo válidos en C#.

Los valores asignados a una variable *carácter* deben ir encerrados entre comillas simples. Por ejemplo, el siguiente ejemplo asigna el valor X a la variable **ch**:

```
char ch;
ch = 'X';
```

Puedes presentar un valor **char** utilizando la declaración **WriteLine()**. Por ejemplo, la siguiente línea de código presenta en pantalla el valor de **ch**:

```
Console.WriteLine("Éste es un char: " + ch);
```

Aunque **char** es definido por C# como un tipo entero, no puede ser mezclado libremente con números enteros en todas las clases. Esto se debe a que no existe una conversión automática de tipos de número entero a **char**. Por ejemplo, el siguiente fragmento es inválido:

```
char ch;

ch = 10; // error, esto no funciona
```

La razón por la que no funcionará el código anterior es porque 10 es un valor entero y no se convertirá automáticamente en **char**. Por ello, la asignación contiene tipos incompatibles. Si intentas compilar este código recibirás un mensaje de error. Más adelante en este mismo capítulo encontrarás una manera de evitar esta restricción.

## Pregunta al experto

**P:** ¿Por qué C# utiliza Unicode?

**R:** C# fue diseñado para permitir la escritura de programas aplicables en todo el mundo. Por lo mismo, necesita utilizar un conjunto de caracteres que represente a todos los idiomas del mundo. Unicode es el conjunto estándar de caracteres diseñado específicamente con ese propósito. Por supuesto, el uso de Unicode es innecesario para lenguajes como el inglés, alemán, español o francés, cuyos caracteres están contenidos en 8 bits, pero ése es el precio de la portabilidad global.

## El tipo **bool**

El tipo **bool** representa valores verdadero/falso. C# define estos valores utilizando las palabras reservadas **true** y **false**, respectivamente. Así, una variable o expresión tipo **bool** contendrá sólo uno de esos dos valores. Más aún, no hay un método definido para la conversión entre valores **bool** y enteros. Por ejemplo, 1 no se convierte en verdadero, ni 0 se convierte en falso.

A continuación presentamos un programa que muestra el funcionamiento del tipo **bool**:

```
// Muestra valores bool.

using System;

class BoolDemo {
    static void Main() {
        bool b;

        b = false;
        Console.WriteLine("b es " + b);
        b = true;
        Console.WriteLine("ahora b es " + b);

        // Un valor bool puede controlar una declaración if.
        if (b) Console.WriteLine("Esto se ejecuta.");
        ↑
        b = false;
        if (b) Console.WriteLine("Esto no se ejecuta.");

        // La salida de un operador relacional es un valor bool.
        Console.WriteLine("88 > 17 es " + (88 > 17));
    }
}
```

Un solo valor **bool**  
puede controlar una  
declaración **if**.

Los datos de salida generados por este programa son los siguientes:

```
b es False
ahora b es True
Esto se ejecuta.
88 > 17 es True
```

Hay tres aspectos interesantes a destacar sobre este programa. Primero, como puedes ver, cuando un valor **bool** es presentado por **WriteLine()** el resultado es “True” o “False”, Verdadero o Falso, respectivamente. Segundo, el valor de una variable **bool** es suficiente, por sí mismo, para controlar una declaración **if**. No hay necesidad de escribir la declaración **if** de la manera convencional:

```
if(b == true) ...
```

Tercero, el resultado de un operador relacional, como **<**, es un valor **bool**. Por ello la expresión **88 > 17** da como resultado “True”. El conjunto extra de paréntesis alrededor de la expresión **88 > 17** es necesario porque el operador **+** tiene una precedencia superior al operador **>**.

## Algunas opciones para datos de salida

Antes de continuar con nuestro análisis de tipos de datos y operadores, será útil abrir un pequeño paréntesis. Hasta este momento, para presentar listas de datos has utilizado el signo de suma para separar cada parte de la lista, como se muestra a continuación:

```
Console.WriteLine("Has ordenado " + 2 + " cosas de $" + 3 + " cada una.");
```

Aunque muy conveniente, enviar datos de salida numéricos de esta manera no te da ningún control sobre la forma en que aparece la información. Por ejemplo, en un valor de punto flotante no puedes controlar la cantidad de decimales que se muestran. Como en el siguiente ejemplo:

```
Console.WriteLine("Dividir 10/3 es igual a " + 10.0/3.0);
```

Genera el resultado:

```
Dividir 10/3 es igual a 3.33333333333333
```

Si bien una respuesta similar puede ser útil para algunos propósitos, es posible que sea inapropiada para otros. Por ejemplo, para cálculos financieros sólo querrás desplegar dos decimales.

Para controlar el formato de los datos numéricos, deberás utilizar una forma alternativa de **WriteLine( )**, que se muestra a continuación y que te permite incrustar información sobre el formato:

```
WriteLine("cadena con formato", arg0, arg1, ..., argN)
```

En esta versión, los argumentos (*arg*) para **WriteLine( )** están separados por comas y no por signos de suma. La *cadena con formato* contiene dos elementos: caracteres regulares e imprimibles que se muestran tal como son y especificadores de formato. Los especificadores de formato siguen el siguiente orden general:

```
{argnum, width:fmt}
```

Aquí, *argnum* especifica el número del argumento que será mostrado (comenzando de cero). *With* especifica el espacio mínimo entre un argumento y otro dentro de la línea de presentación. Finalmente, *fmt* indica el formato con que aparecerá el entero.

Durante la ejecución, cuando el programa encuentra un especificador de formato en la cadena, el argumento especificado por *argnum* es sustituido por su valor correspondiente; lo mismo sucede con los otros dos elementos; si se indicó una posición determinada con *width*, el programa colocará el valor de ese argumento en el lugar indicado y si se le asignó cierto formato también ejecutará la orden. Tanto *width* como *fmt* son características opcionales. Así, en su forma más sencilla, un especificador de formato simplemente indica los argumentos que se deben mostrar en la salida. Por ejemplo, {0} indica que se debe mostrar el *arg0*, {1} hace referencia al *arg1*, y así sucesivamente.

Comencemos con un ejemplo sencillo. La declaración

```
Console.WriteLine("Febrero tiene {0} o {1} días.", 28, 29);
```

produce la siguiente salida:

```
Febrero tiene 28 o 29 días.
```

Como puedes ver, {0} es sustituido por 28 y {1} por 29. Los especificadores de formato identifican el lugar donde los respectivos argumentos deben aparecer, en este caso 28 y 29. Advierte además que los valores están separados por comas y no por signos de suma.

A continuación aparece una variante de la anterior declaración que especifica la separación entre argumentos (*width*):

```
Console.WriteLine("Febrero tiene {0,10} o {1,5} días.", 28, 29);
```

Produce la siguiente salida:

```
Febrero tiene      28 o      29 días.
```

Como puedes ver, se han añadido espacios en blanco para llenar las porciones sin uso entre argumentos. Recuerda que el segundo valor dentro de las llaves indica la posición que tendrá el argumento correspondiente dentro de la línea de salida. Los datos que se muestran pueden exceder ese tamaño entre argumentos de ser necesario.

En los ejemplos anteriores no se aplicó ningún formato a los valores en sí. Por supuesto, el valor de utilizar especificadores de formato es controlar la apariencia de los datos de salida. Los tipos de datos que son formateados con mayor frecuencia son los de punto flotante y los decimales. Una de las maneras más fáciles para especificar el formato es especificar el modelo que utilizará **WriteLine()**. Para hacerlo, muestra un ejemplo del formato que quieras aplicar utilizando el símbolo # para marcar la posición de los dígitos. Por ejemplo, he aquí una mejor manera de presentar el resultado de 10 dividido entre 3:

```
Console.WriteLine("Dividir 10/3 es igual a {0:#.##}", 10.0/3.0);
```

El resultado que se genera ahora es:

```
Dividir 10/3 es igual a 3.33
```

En este ejemplo el modelo es **#.##**, el cual le dice a **WriteLine()** que muestre dos posiciones decimales. Es importante comprender, sin embargo, que **WriteLine()** mostrará más de un dígito del lado izquierdo del punto en caso necesario, para no falsear el resultado correcto.

Si quieras mostrar valores monetarios, debes utilizar el especificador de formato **C**, como en el siguiente ejemplo:

```
decimal balance;  
  
balance = 12323.09m;  
Console.Writeline("El balance actual es {0:C}", balance);
```

El resultado de esta línea de código es el siguiente (en formato de dólares norteamericanos):

```
El balance actual es $12,323.09
```

## Prueba esto

## Hablar a Marte

En su punto más cercano a la Tierra, Marte se encuentra a 55 millones de kilómetros aproximadamente. Suponiendo que en Marte hay alguien con quien te quieras comunicar, ¿cuál sería el retraso de la señal de radio desde el instante en que abandona la Tierra hasta el momento en que llega a Marte? El siguiente programa proporciona la respuesta. Recuerda que la luz viaja a 300 000 kilómetros por segundo, aproximadamente. Para calcular el retraso necesitarás dividir la distancia entre la velocidad de la luz. Debes mostrar el retraso en formato de minutos y segundos.

(continúa)

## Paso a paso

1. Crea un nuevo archivo llamado **Marte.cs**.
2. Para calcular el retraso necesitarás valores de punto flotante. ¿Por qué? Porque el intervalo tendrá un componente fraccional. He aquí las variables que se utilizarán en el programa:

```
double distancia;
double velocidadluz;
double retraso;
double retraso_en_min;
```

3. Los valores para la **distancia** y la **velocidadluz** son:

```
distancia = 55000000 // 55 millones de kilómetros
velocidadluz = 300000 // 300 mil kilómetros por segundo
```

4. Para calcular el retraso divide la **distancia** entre la **velocidadluz**. El resultado es el retraso expresado en segundos. Asigna este valor a la variable **retraso** y muestra el resultado. Los pasos son los siguientes:

```
retraso = distancia / velocidadluz;

Console.WriteLine("Tiempo de retraso cuando se habla a Marte: " +
    retraso + " segundos.");
```

5. Divide la cantidad de segundos en **retraso** entre 60 para obtener los minutos; muestra el resultado utilizando la siguiente línea de código:

```
retraso_en_min = retraso / 60;

Console.WriteLine("Que son " + retraso_en_min +
    " minutos.");
```

Éste es el programa Marte.cs completo:

```
// Llamar a Marte

using System;

class Marte {
    static void Main() {
        double distancia;
        double velocidadluz;
        double retraso;
        double retraso_en_min;

        distancia = 55000000; // 55 millones de kilómetros
        velocidadluz = 300000; // 300 mil kilómetros por segundo

        retraso = distancia / velocidadluz;
```

```
Console.WriteLine("Tiempo de retraso cuando se habla a Marte: " +
    retraso + " segundos.");
    retraso_en_min = retraso / 60;
    Console.WriteLine("Que son " + retraso_en_min +
        " minutos.");
}
}
```

6. Compila y ejecuta el programa. Se muestra el siguiente resultado:

```
Tiempo de retraso cuando se habla a Marte: 183.333333333333 segundos.
Que son 3.05555555555556 minutos.
```

7. Mucha gente pensará que muestra demasiados decimales. Para mejorar la legibilidad del resultado, sustituye las declaraciones **WriteLine()** del programa con las que aparecen a continuación:

```
Console.Writeline("Tiempo de retraso cuando se habla a Marte: {0:#.###} 
    segundos", retraso);
Console.Writeline("Que son: {0:#.###} minutos", retraso_en_min);
```

8. Vuelve a compilar y ejecutar el programa. Cuando lo hagas verás el siguiente resultado:

```
Tiempo de retraso cuando se habla a Marte: 183.333 segundos
Que son: 3.056 minutos
```

Ahora sólo aparecen tres decimales.

---

## Literales

En C# el término *literales* se refiere a valores fijos que son representados en su forma comprensible para los humanos. Por ejemplo, el número 100 es una literal. Las literales y su uso son tan intuitivos que ya las hemos utilizado de una manera u otra en todos los ejemplos anteriores. Ahora es momento de explicarlas formalmente.

Las literales de C# pueden representar cualquier valor de todos los tipos de datos. La forma de representar cada literal depende del tipo de dato al que pertenece. Como se explicó con anterioridad, las literales de carácter van encerradas entre comillas simples, por ejemplo 'a' y '%' son literales de carácter.

Las literales de entero son especificadas como números sin componentes decimales. Por ejemplo, 10 y -100 son literales de entero. Las literales de punto flotante requieren el uso del punto decimal seguido por los decimales correspondientes. Por ejemplo, 11.123 es una literal de punto flotante.

Como C# es un lenguaje estrictamente tipificado, las literales también tienen un tipo. Como es natural, esto hace surgir una duda: ¿cuál es el tipo de una literal numérica? Por ejemplo, ¿cuál es el tipo de 12, 123987 o 0.23? Por fortuna, C# especifica una serie de reglas fáciles de seguir que responden tales cuestionamientos.

Primero, para literales de enteros, el tipo de la literal es el tipo de entero más pequeño que puede contenerla, comenzando con **int**. De esta manera, una literal de entero es de tipo **int**, **uint**, **long** o **ulong**, dependiendo del valor que represente. Segundo, las literales de punto flotante son del tipo **double**.

Si el tipo asignado en automático por C# no es el que requieres en una literal, puedes especificar explícitamente un nuevo tipo incluyendo el sufijo correspondiente. Para especificar una literal tipo **long** añade al final una *l* o *L*. Por ejemplo, 12 es un **int** por defecto, pero puedes escribir 12L para convertirla en un tipo **long**. Para especificar un valor entero sin negativos, añade *u* o *U*. De esta manera 100 es un **int**, pero 100U es un **uint**. Para especificar un entero largo sin negativos utiliza *ul* o *UL*. Por ejemplo, 984375UL es de tipo **ulong**.

Para especificar una literal **float**, añade *f* o *F*. Por ejemplo, 10.19F es de tipo **float**. Aunque es una redundancia, puedes especificar una literal **double** añadiendo **d** o **D**. Como acabo de mencionar, las literales de punto flotante son de tipo **double** por defecto.

Para especificar una literal **decimal**, añade al valor una *m* o *M*. Por ejemplo, 9.95M es una literal **decimal**.

Aunque las literales de entero crean valores **int**, **uint**, **long** y **ulong** por defecto, también pueden ser asignadas a variables tipo **byte**, **sbyte**, **short** y **ushort**, siempre y cuando el valor asignado pueda ser representado por el tipo que lo contiene.

## Literales hexadecimales

En programación, a veces es más fácil utilizar un sistema numérico base 16 en lugar de la base 10. El sistema numérico base 16 es llamado *hexadecimal* y utiliza los dígitos del 0 al 9, más las letras de la A a la F, que representan 10, 11, 12, 13, 14 y 15. Por ejemplo, la representación hexadecimal del número 10 es 16. Dada la frecuencia con que son utilizados los números hexadecimales, C# te permite especificar literales de entero con formato hexadecimal. Una literal hexadecimal debe comenzar con 0x (un cero seguido por una equis minúscula, x). He aquí algunos ejemplos:

```
cuenta = 0xFF; // 255 en el sistema decimal  
incremento = 0x1a; // 26 en el sistema decimal
```

## Secuencias de caracteres de escape

Encerrar caracteres entre comillas simples funciona para la mayoría de las grafías imprimibles, pero algunos caracteres especiales como los saltos de línea presentan algunos problemas cuando se utiliza un editor de textos. Además, algunos otros caracteres, como las comillas simples y dobles tienen un significado especial en C#, por lo que no puedes utilizarlos directamente. Por esa razón, C# proporciona las secuencias de caracteres de escape que se muestran en la tabla 2-2. Estas secuencias son utilizadas para reemplazar los caracteres que representan y que no pueden ser utilizados directamente en el código.

Por ejemplo, para asignar a la variable **ch** el carácter de tabulación se utiliza el siguiente código:

```
ch = '\t';
```

El siguiente ejemplo asigna una comilla simple a **ch**:

```
ch = '\'';
```

Secuencia de escape	Descripción
\a	Alerta (campana)
\b	Tecla de retroceso
\f	Salto de hoja
\n	Nueva línea (salto de línea)
\r	Regreso de carro
\t	Tabulador horizontal
\v	Tabulador vertical
\0	Nulo
'	Comilla simple
"	Comilla doble
\\"	Diagonal invertida

**Tabla 2-2** Secuencias de caracteres de escape

## Literales de cadena de caracteres

C# soporta otro tipo de literales: las cadenas de caracteres o simplemente cadenas. Una literal de cadena es un conjunto de caracteres encerrados entre comillas dobles. Por ejemplo:

"Ésta es una prueba"

es una cadena de caracteres. Has visto ejemplos de ellas en muchas de las declaraciones **WriteLine()** en los programas de ejemplos anteriores.

Además de caracteres normales, una literal de cadena también contiene una o más de las secuencias de escape que acabamos de describir. A manera de ejemplo analiza el siguiente programa; utiliza las secuencias de escape \n y \t.

## Pregunta al experto

**P:** Sé que C++ permite que las literales de entero sean especificadas en formato octal (un sistema numérico base 8). ¿C# permite el uso de literales octales?

**R:** No. C# sólo permite la especificación de literales de entero con formato decimal y hexadecimal. El formato octal es raramente utilizado en los ambientes de programación modernos.

## 54 Fundamentos de C# 3.0

```
// Muestra secuencias de escape en cadenas de caracteres.  
using System;  
  
class StrDemo {  
    static void Main() {  
        Console.WriteLine("Primera línea\nSegunda línea");  
        Console.WriteLine("A\tB\tC");  
        Console.WriteLine("D\tE\tF");  
    }  
}
```

Usa `\n` para generar una línea nueva.

← Usa tabulaciones para alinear el resultado.

Los datos de salida son los siguientes:

```
Primera línea  
Segunda línea  
A      B      C  
D      E      F
```

Advierte cómo se utiliza la secuencia de escape `\n` para generar un salto de línea. No necesitas utilizar varias declaraciones `WriteLine()` para obtener datos de salida en diversas líneas, simplemente debes incrustar `\n` dentro de una cadena larga, en el punto donde quieras que comience la nueva línea.

Además del formato para literales de cadena que acabamos de describir, también puedes especificar una *literal de cadena verbatim*.<sup>1</sup> Para aplicar este formato debes comenzar con una arroba (@), seguida por la cadena de caracteres encerrada entre comillas dobles. El contenido de la cadena es aceptado sin modificaciones y puede abarcar dos o más líneas, tabuladores y demás, pero no es necesario que utilices secuencias de escape. La única excepción son las comillas dobles, las cuales debes utilizar en secuencia y sin separación (""). A continuación presentamos un programa que utiliza literales de cadena verbatim:

```
// Muestra literales de cadena verbatim.  
using System;  
  
class Verbatim {  
    static void Main() {  
        Console.WriteLine(@"Ésta es una literal  
de cadena verbatim  
que abarca varias líneas.  
");  
  
        Console.WriteLine(@"He aquí un ejemplo de tabulación:  
1      2      3      4  
5      6      7      8  
");  
  
        Console.WriteLine(@"Los programadores dicen: ""Me gusta C#. """ );  
    }  
}
```

← Este comentario verbatim tiene líneas incrustadas.

← Ésta también contiene tabulaciones incrustadas.

<sup>1</sup> *Verbatim* es una palabra en latín que significa “al pie de la letra”.

Los datos de salida de este programa son:

```
Ésta es una literal  
de cadena verbatim  
que abarca varias líneas.
```

He aquí un ejemplo de tabulación:

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

Los programadores dicen: "Me gusta C#."

El punto importante a destacar del programa anterior es que las literales de cadena verbatim se muestran exactamente como son escritas en el código del programa.

La ventaja del formato verbatim es que puedes especificar los datos de salida de tu programa desde el código exactamente como aparecerá en la pantalla. Sin embargo, en el caso de múltiples líneas, los saltos pueden hacer que la indentación de tu programa sea confusa. Por esta razón, los programas de este libro harán un uso limitado de las literales de cadena verbatim. Dicho esto, podemos afirmar que son de enorme ayuda para muchas situaciones donde se requiere un formato específico.

## Un análisis profundo de las variables

En el capítulo 1 presentamos una introducción a las variables. Como ya lo sabes, las variables se declaran utilizando el formato declarativo:

*tipo nombre-variable;*

donde *tipo* es el tipo de dato asignado a la variable y *nombre-variable* es el nombre distintivo de esta pieza del programa. Puedes declarar variables de cualquier tipo válido, incluyendo los descritos en páginas anteriores. Es importante comprender que las capacidades de la variable están determinadas por el tipo de dato que les asignas. Por ejemplo, una variable de tipo **bool** no puede ser utilizada para contener valores de punto flotante. Más aún, el tipo de la variable no puede cambiarse durante su tiempo de vida: una variable **int** no puede transformarse en **char**, por ejemplo.

Todas las variables en C# deben declararse. Esto es necesario porque el compilador debe saber qué tipo de dato contiene una variable antes de poder compilar apropiadamente cualquier declaración que la utilice. Esto permite también que C# aplique una verificación estricta de los tipos de dato.

C# define diferentes especies de variables. Las que hemos utilizado hasta ahora reciben el nombre de *variables locales*, porque se declaran dentro de un método.

### Pregunta al experto

**P:** ¿Una cadena de caracteres compuesta por un solo elemento es lo mismo que una literal de carácter? Por ejemplo, ¿"k" es lo mismo que 'k'?

**R:** No. No debes confundir las cadenas con las literales de carácter. Una literal de carácter representa una sola letra de tipo **char**. Una cadena que contiene una sola letra no deja de ser una cadena. Aunque las cadenas están conformadas por caracteres, no son del mismo tipo.

## Inicializar una variable

Una manera de darle cierto valor a una variable es a través de la declaración de asignación, como ya lo has visto. Otra manera es darle un valor inicial cuando es declarada. Para hacerlo, se escribe un signo de igual después del nombre de la variable y a continuación el valor que le será asignado. El formato general de la inicialización es el siguiente:

*tipo nombre-variable = valor;*

Aquí, *valor* es la valía asignada a *nombre-variable* cuando ésta es creada. El *valor* debe ser compatible con el *tipo* especificado al inicio.

He aquí algunos ejemplos:

```
int cuenta = 10; // otorga a la variable cuenta un valor inicial de 10
char ch = 'X'; // inicializa la variable ch con la letra X
float f = 1.2F; // la variable f se inicializa con un valor de 1.2
```

Cuando se declaran dos o más variables del mismo tipo a manera de lista y utilizando comas para separarlas, también es posible asignarles un valor inicial. Por ejemplo:

```
int a, b = 8, c = 19, d; // b y c han sido inicializadas
```

En este caso, sólo las variables **b** y **c** son inicializadas.

## Inicialización dinámica

Aunque en los ejemplos anteriores se han utilizado constantes como inicializadores, C# permite que una variable sea inicializada dinámicamente, utilizando cualquier expresión válida en el punto donde la variable es declarada. Por ejemplo, a continuación tenemos un pequeño programa que calcula el volumen de un cilindro a parir del radio de su base y su altura.

```
// Demuestra la inicialización dinámica.

using System;

class InicioDinámico {
    static void Main() {
        double radio = 4, alto = 5;
        // Inicializa el volumen dinámicamente.
        double volumen = 3.1416 * radio * radio * alto;
        Console.WriteLine("El volumen es " + volumen);
    }
}
```

**volumen** se inicia dinámicamente  
al tiempo de ejecución.

En el ejemplo se declaran las tres variables (**radio**, **alto** y **volumen**). Las dos primeras, **radio** y **alto**, son inicializadas con constantes. Pero **volumen** es inicializada dinámicamente con la relación de las dos anteriores que proporciona el volumen del cilindro. El punto clave del ejemplo es que las expresiones de inicialización pueden utilizar cualquier elemento válido en el punto de la inicialización, incluyendo invocaciones a métodos, otras variables o literales.

## Variables de tipo implícito

Como ya se explicó, en C# las variables deben ser declaradas. Normalmente la declaración incluye el tipo de la variable, como **int**, o **bool**, seguido de su nombre. Sin embargo, en C# es posible dejar que el compilador determine el tipo de variable basándose en el valor que se utilizó para inicializarla. A esto se llama *variables de tipo implícito*.

Una variable de tipo implícito es declarada utilizando la palabra clave **var** y también debe ser inicializada. El compilador utiliza el tipo del inicializador para determinar el tipo de dato que le corresponde. He aquí un ejemplo:

```
var pi = 3.1416;
```

Como **pi** se inicializa con una literal de punto flotante (cuyo tipo por omisión es **double**), el tipo de **pi** es **double**. Si **pi** se declarara de la siguiente manera:

```
var pi = 3.1416M;
```

el tipo de **pi** sería **decimal**, tal y como lo indica la M.

El siguiente programa muestra el uso de las variables de tipo implícito:

```
// Muestra variables de tipo implícito.

using System;

class VarTipImpl {
    static void Main() {

        // Son variables de tipo implícito.
        var pi = 3.1416; // pi es tipo double
        var radio = 10; // radio es de tipo int
        // msg y msg2 son variables tipo cadena (string).
        var msg = "Radio: ";
        var msg2 = "Área: ";

        // Declaración explícita de área como double.
        double area;

        Console.WriteLine(msg2 + radio);
        area = pi * radio * radio;
        Console.WriteLine(msg + area);

        Console.WriteLine();

        radio = radio + 2;

        Console.WriteLine(msg2 + radio);
        area = pi * radio * radio;
```

The diagram shows a callout box with the text "Variables de tipo implícito." pointing to the four variable declarations at the top of the code block. The declarations are highlighted with a light gray background.

```
Console.WriteLine(msg + area);

// La siguiente declaración no se compilará porque
// radio es un int y no puede ser asignado a un valor
// de punto flotante.
//     radio = 12.2; // Error!
}

}
```

Los datos de salida generados por este programa son:

```
Área: 10
Radio: 314.16

Área: 12
Radio: 452.3904
```

Es importante enfatizar que una variable de tipo implícito sigue siendo estrictamente tipificada. Pon atención a la última línea del programa que está comentada para que no se ejecute:

```
//     radio = 12.2; // Error!
```

Esta asignación es inválida porque **radio** es de tipo **int**, razón por la que no es posible asignarle el tipo **double**. La única diferencia entre las variables de tipo implícito y las variables “normales” de tipo explícito, es el mecanismo a través del cual se determina el tipo de dato al que pertenecen. Una vez que el tipo ha sido determinado, la variable pertenece a ese tipo específico y no es posible cambiarlo durante su periodo de vida. Así, el tipo de **radio** no puede cambiar durante la ejecución del programa.

Las variables de tipo implícito fueron incorporadas a C# para manejar situaciones y casos especiales; las más importantes de ellas están relacionadas con LINQ (consulta de lenguaje integrado), que se aborda más tarde en este libro. Para la mayoría de las variables debes utilizar el tipo explícito porque facilitan la lectura y comprensión de tu código. Las variables de tipo implícito sólo deben utilizarse cuando sea necesario. No fueron creadas con la intención de sustituir las declaraciones estándar de variable. En pocas palabras, debes usar y no abusar de esta nueva característica de C#.

Un último punto: sólo es posible declarar una variable de tipo implícito a la vez. Por lo que la declaración

```
var cuenta = 10, max = 20; // Error!
```

es incorrecta y no compilará porque intenta declarar **cuenta** y **max** al mismo tiempo.

## El alcance y tiempo de vida de las variables

Hasta el momento, todas las variables que has usado se han declarado al principio del método **Main()**, pero C# permite que una variable local sea declarada dentro de cualquier bloque. Como se explicó en el capítulo 1, un bloque de código inicia con una llave de apertura (**{**) y finaliza con una llave de clausura (**}**). Un bloque define cierto *alcance*, por lo que cada vez que inicias un nuevo bloque estás creando una nueva extensión de código que determina cuáles variables son visibles para otras partes de tu programa. También determina el tiempo de vida de tus variables locales.

Los alcances más importantes en C# son aquellos definidos por una clase y los que define un método. La explicación sobre el alcance de una clase (y las variables declaradas en ella) se verá más tarde en este libro, cuando las clases se describan con detalle. En este momento nos concentraremos en los alcances y extensiones definidas por y dentro de un método.

La extensión definida por un método comienza con la llave de apertura y finaliza con la llave de clausura. Si ese método tiene parámetros, ellos también están limitados a la extensión del método que los contiene.

Como regla general, las variables locales declaradas dentro de cierto bloque no son visibles para el código definido fuera de ese bloque. Así, cuando declaras una variable dentro de cierto bloque, estás previniendo que sea utilizada o modificada por código que esté fuera de su demarcación. En efecto, las reglas de los alcances son el fundamento del encapsulado.

Los bloques de código (y por tanto sus alcances) pueden anidarse. Por ejemplo, cada vez que creas un nuevo bloque de código estás creando una nueva extensión anidada. Cuando esto sucede, la extensión exterior encierra a la interior. Esto significa que las variables declaradas en la extensión exterior pueden ser vistas por las variables de la extensión interior, pero no a la inversa. Las variables declaradas dentro de la extensión interior no pueden ser vistas fuera de su demarcación.

Para comprender el efecto de los alcances anidados, examina el siguiente programa:

```
// Muestra el alcance de los bloques.

using System;

class AlcanceDemo {
    static void Main() {
        int x; // variable vista por todo el código dentro de Main()

        x = 10;
        if(x == 10) { // inicia una nueva extensión
            int y = 20; // variable vista sólo dentro de este bloque

            // Tanto x como y son visibles en este punto.
            Console.WriteLine("x y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Error! y no es visible en este punto ← En este punto, la
        // x todavía es visible aquí.                                variable y está fuera
        // de su demarcación.
        Console.WriteLine("x es " + x);
    }
}
```

Como se indica en los comentarios, la variable **x** está declarada al inicio de **Main()** y por ello tiene acceso a todo el código dentro de su extensión. Por su parte, **y** está declarada dentro del bloque que pertenece a **if**, dado que el bloque define el alcance, y sólo es visible por el código que se encuentra dentro del mismo bloque. Por esa razón, la línea **y = 100;** está fuera de su demarcación y se encuentra comentada; si eliminas la doble diagonal del comentario se presentará un error de compilación, porque **y** no es visible fuera de su bloque. Sin embargo, **x** puede utilizarse dentro del bloque **if** porque se trata de un bloque interior (es decir, una extensión anidada) y el código declarado en los bloques exteriores tiene acceso a los interiores.

Las variables locales pueden declararse en cualquier punto dentro del bloque, pero son útiles sólo después de su declaración. En este sentido, si defines una variable al principio de un método, estará disponible para todo el código que se encuentra dentro de ese método. Pero, por el contrario, si declaras una variable al final del bloque, será completamente inútil porque ninguna parte del código tendrá acceso a ella.

Si la declaración de variable incluye un inicializador, la variable será reinicializada cada vez que se ejecute el bloque donde está declarada. Analiza el siguiente programa como ejemplo:

```
// Muestra el tiempo de vida de una variable.

using System;

class VarInitDemo {
    static void Main() {
        int x;

        for(x = 0; x < 3; x++) {
            int y = -1; // y se inicializa cada vez que el bloque es ejecutado
            Console.WriteLine("y es: " + y); // esto siempre presenta el
            valor -1
            y = 100;
            Console.WriteLine("y es ahora: " + y);
        }
    }
}
```

Los datos de salida que genera este programa son:

```
y es: -1
y es ahora: 100
y es: -1
y es ahora: 100
y es: -1
y es ahora: 100
```

Como puedes ver, **y** es siempre reinicializada con el valor **-1** cada vez que se ejecuta el bloque **for**. Aunque posteriormente se le asigne el valor **100**, este último se pierde.

Hay una rareza en las reglas de alcance de C# que tal vez te sorprenda: aunque los bloques puedan anidarse, ninguna variable declarada en alguno de los bloques internos puede tener el mismo nombre que una declarada en algún bloque externo. Tenemos como ejemplo el siguiente programa, que intenta declarar dos variables diferentes con el mismo nombre y no se compilará:

```
/*
Este programa intenta declarar una variable
en un bloque interno con el mismo nombre de
una definida en un bloque externo.

*** Este programa no se compilará. ***
*/
```

```
using System;
```

```
class VariableAnidada {  
    static void Main() {  
        int cuenta;  
  
        for(cuenta = 0; cuenta < 10; cuenta = cuenta+1) {  
            Console.WriteLine("Ésta es cuenta: " + cuenta);  
  
            // Ilegal!!! Entra en conflicto con la variable anterior llamada  
            // cuenta.  
            int cuenta; ← No es posible declarar cuenta otra vez,  
            // porque ya ha sido declarada en Main()  
            for(cuenta = 0; cuenta < 2; cuenta++)  
                Console.WriteLine("Este programa es erróneo! ");  
        }  
    }  
}
```

Si tienes experiencia en C/C++, sabes que no hay restricciones para nombrar variables declaradas dentro de bloques internos. En aquellos lenguajes, la declaración de la variable **cuenta** en un bloque exterior al loop **for** es completamente válida. Sin embargo, en C/C++ tal declaración oculta la variable exterior **cuenta**. Los diseñadores de C# pensaron que este tipo de *nombres ocultos* podrían conducir fácilmente a cometer errores de programación y los desecharon.

## Operadores

C# proporciona un rico ambiente de operadores. Un operador es un símbolo que ordena al compilador la realización de una manipulación matemática o lógica específica. C# cuenta con cuatro clases generales de operadores: *aritméticos*, *bitwise*, *relacionales* y *lógicos*. También tiene otros varios operadores que pueden manejar ciertas situaciones especiales. En este capítulo estudiaremos los operadores aritméticos, relacionales y lógicos. También veremos el operador de asignación. Los operadores especiales, incluyendo bitwise, serán estudiados más tarde.

## Operadores aritméticos

C# define los siguientes operadores aritméticos:

Operador	Significado
+	Suma
-	Resta (también signo negativo)
*	Multiplicación
/	División
%	Módulo
++	Incremento
--	Decremento

Los operadores +, -, \* y / funcionan de la manera tradicional, tal y como era de esperarse. Pueden aplicarse a cualquier tipo de dato numérico.

Aunque las acciones de los operadores aritméticos básicos son bien conocidas por todos los lectores, el operador % requiere cierta explicación. Primero, recuerda que cuando se aplica / a un entero, todo remanente será truncado; por ejemplo, 10/3 es igual a 3 en una división de enteros. Puedes obtener el remanente de esta división utilizando el operador módulo (%), que funciona en C# de la misma manera como funciona en los demás lenguajes de cómputo: obtiene el remanente de una división entre enteros. Por ejemplo, 10%3 es igual a 1. En C# el módulo puede aplicarse a valores enteros y de punto flotante. Así, 10.0%3.0 es también igual a 1. En esto difiere de C/C++ que sólo acepta el uso de módulo en valores enteros, tipo **int**. El siguiente programa muestra el uso del operador módulo:

```
// Muestra el uso del operador módulo.

using System;

class ModDemo {
    static void Main() {
        int irest, irem;
        double drest, drem;

        irest = 10 / 3;
        irem = 10 % 3; ←
        drest = 10.0 / 3.0;
        drem = 10.0 % 3.0; ← Uso del operador módulo.

        Console.WriteLine("Resultado y remanente de 10 / 3: " +
                           irest + " " + irem);
        Console.WriteLine("Resultado y remanente de 10.0 / 3.0: " +
                           drest + " " + drem);
    }
}
```

Los datos de salida del programa son:

```
Resultado y remanente de 10 / 3: 3 1
Resultado y remanente de 10.0 / 3.0: 3.33333333333333 1
```

Como puedes ver, el operador % presenta el remanente 1 tanto para la operación con enteros como para la de punto flotante.

## Incremento y decremento

Los operadores de incremento (++) y decremento (--) fueron presentados en el capítulo 1. Como verás, tienen algunas propiedades especiales que los hacen muy interesantes. Comencemos por recordar precisamente lo que hace cada uno de estos operadores.

El operador de incremento suma 1 a su operando, y el operador de decremento le resta 1. Por tanto:

```
x = x + 1;
```

es lo mismo que

```
x++;
```

y

```
x = x - 1;
```

es lo mismo que

```
--x;
```

Tanto el operador de incremento como el de decremento pueden anteceder al operador (*prefijo*) o bien sucederlo (*sufijo*). Por ejemplo:

```
x = x + 1;
```

puede escribirse como

```
++x; // formato de prefijo
```

o como

```
x++ // formato de sufijo
```

En el ejemplo anterior, no hay diferencia si el incremento es aplicado como prefijo o como sufijo. Sin embargo, cuando el incremento o el decremento son aplicados como parte de una expresión larga, sí hay una diferencia importante. En este caso, cuando el operador *antecede* al operando, el resultado de la operación es el valor del operando *después* del incremento o decremento. Si el operador *sigue* a su operando, el resultado de la operación es el valor del operando *después* del incremento o decremento. Analiza el siguiente código:

```
x = 10;  
y = ++x;
```

En este caso el valor de **y** será 11, porque primero se incrementa **x** y luego se obtiene su valor. Sin embargo, si el código se escribiera de la siguiente manera:

```
x = 10;  
y = x++;
```

entonces el valor de **y** sería 10. En este caso, primero se obtiene el valor de **x**, luego se incrementa y después el valor original de **x** es devuelto. En ambos casos, **x** sigue siendo 11; la diferencia radica en el orden que utiliza el operador.

He aquí un ejemplo poco más complejo:

```
x = 10;  
y = 2;  
z = x++ - (x * y);
```

En este caso el valor de **z** es -12. Porque cuando **x++** es evaluada, el valor de **x** es 11, pero transmite el valor 10 (el valor original de **x**). Esto significa que a **z** se le asigna el valor  $10 - (11 * 2)$ , que es -12. Sin embargo, cuando se escribe así:

```
z = ++x - (x * y);
```

El resultado es -11, porque primero se incrementa **x** y luego se obtiene su valor. Así, a **z** se le asigna el valor  $11 - (11 * 2)$ , igual a -11. Como lo muestra este ejemplo, la posibilidad de controlar el orden de las operaciones de incremento y decremento puede tener ventajas significativas.

## Operadores lógicos y de relación

En las expresiones *operador relacional* y *operador lógico*, el término *relacional* hace referencia a las mutuas relaciones que pueden darse entre diferentes valores, y el término *lógico* se refiere a las reglas de interconexión entre los valores verdadero y falso. Como los operadores relacionales producen valores verdadero y falso, suelen trabajar en conjunto con los operadores lógicos. Por tal razón, los analizaremos juntos en esta sección.

Los operadores relacionales son los siguientes:

Operador	Significado
<b>==</b>	Igual a
<b>!=</b>	Diferente de
<b>&gt;</b>	Mayor que
<b>&lt;</b>	Menor que
<b>&gt;=</b>	Mayor que o igual a
<b>&lt;=</b>	Menor que o igual a

Los operadores lógicos se muestran a continuación:

Operador	Significado
<b>&amp;</b>	AND (Y)
<b> </b>	OR (O)
<b>^</b>	XOR (OR exclusivo)
<b>  </b>	Circuito corto de OR
<b>&amp;&amp;</b>	Circuito corto de AND
<b>!</b>	NOT (NO)

El resultado que presentan los operadores relacionales y lógicos es un valor tipo **bool**.

En general, los objetos pueden ser comparados por su semejanza o diferencia utilizando **==** y **!=**. Sin embargo, los operadores de comparación **<**, **>**, **<=**, **>=**, sólo pueden aplicarse a los tipos que soportan una relación ordenada. Por tanto, todos los operadores relacionales pueden aplicarse a todos los tipos numéricos. Sin embargo, los valores tipo **bool** sólo pueden ser comparados para obtener similitud o diferencia porque los valores **true** (verdadero) y **false** (falso) no tienen un orden predeterminado. Por ejemplo, **true > false** no tiene significado en C#.

En lo que respecta a los operadores lógicos, los operandos deben ser de tipo **bool** y el resultado de una operación lógica es también de este tipo. Los operadores lógicos **&**, **|**, **^** y **!** soportan las operaciones lógicas básicas dictadas por AND, OR, XOR y NOT de acuerdo con la siguiente tabla de verdad:

<b>p</b>	<b>q</b>	<b>p &amp; q</b>	<b>p   q</b>	<b>p ^ q</b>	<b>!p</b>
Falso	Falso	Falso	Falso	Falso	Verdadero
Verdadero	Falso	Falso	Verdadero	Verdadero	Falso
Falso	Verdadero	Falso	Verdadero	Verdadero	Verdadero
Verdadero	Verdadero	Verdadero	Verdadero	Falso	Falso

Como se muestra en la tabla, el resultado de una operación OR exclusiva es verdadero sólo cuando exactamente un operando es verdadero.

A continuación presentamos un programa que muestra el funcionamiento de varios operadores relacionales y lógicos:

```
// Muestra el funcionamiento de los operadores relacionales y lógicos.
using System;
class RelLogOps {
    static void Main() {
        int i, j;
        bool b1, b2;
        i = 10;
        j = 11;
        if(i < j) Console.WriteLine("i < j");
        if(i <= j) Console.WriteLine("i <= j");
        if(i != j) Console.WriteLine("i != j");
        if(i == j) Console.WriteLine("esto no se ejecutará");
        if(i >= j) Console.WriteLine("esto no se ejecutará");
        if(i > j) Console.WriteLine("esto no se ejecutará");

        b1 = true;
        b2 = false;
        if(b1 & b2) Console.WriteLine("esto no se ejecutará");
        if(!(b1 & b2)) Console.WriteLine("!(b1 & b2) es verdadero");
        if(b1 | b2) Console.WriteLine("b1 | b2 es verdadero");
        if(b1 ^ b2) Console.WriteLine("b1 ^ b2 es verdadero");
    }
}
```

Los datos de salida generados por este programa son los siguientes:

```
i < j
i <= j
i != j
!(b1 & b2) es verdadero
b1 | b2 es verdadero
b1 ^ b2 es verdadero
```

## Operadores lógicos de circuito corto

C# proporciona versiones especiales de los operadores lógicos AND y OR, llamados *circuitos cortos*, que pueden ser utilizados para producir código más eficiente. Para comprender su razón de ser, considera el siguiente razonamiento: en la operación AND, si el primer operando es falso el resultado es falso, sin importar el valor del segundo operando. En la operación OR, si el primer operando es verdadero el resultado es verdadero, sin importar el valor del segundo operando. Así, en ambos casos no es necesario evaluar el segundo operando. Al dejar de evaluar el segundo operando, se ahorra tiempo y se produce código más eficiente.

El circuito corto para el operador AND está representado por `&&`, y para el operador OR por `||`. Como se mencionó anteriormente, sus contrapartes normales son `&` y `|`. La única diferencia entre las versiones normales y los circuitos cortos es que las primeras siempre evaluarán los dos operandos, pero los circuitos cortos sólo evalúan el segundo operando cuando es necesario.

A continuación presentamos un programa que muestra el funcionamiento del circuito corto del operador AND. El programa determina si el valor de `d` es factor de `n`. Para hacerlo realiza una operación de módulo. Si el remanente de `n/d` es igual a cero, entonces `d` es factor. Sin embargo, como la operación de módulo involucra una división, el formato de circuito corto de AND es utilizado para evitar un error proveniente de la división entre cero.

```
// Muestra los operadores de circuito corto.

using System;

class SCops {
    static void Main() {
        int n, d;

        n = 10;
        d = 2;

        // Aquí, d es 2, por lo que aplica la operación de módulo.
        if(d != 0 && (n % d) == 0)
            Console.WriteLine(d + " es un factor de " + n);

        d = 0; // Ahora el valor de d es igual a 0

        // Como d es igual a cero, el segundo operando no se evalúa.
        if (d != 0 && (n % d) == 0) ← El operador de circuito corto previene la división entre cero.
            Console.WriteLine(d + " es un factor de " + n);

        // Ahora, intenta hacer lo mismo sin el operador de circuito corto.
        // Esto provocará un error de división entre cero.
        if (d != 0 & (n % d) == 0) ←
            Console.WriteLine(d + " es un factor de " + n);
    }
}
```

Ahora, ambas expresiones son evaluadas, permitiendo que ocurra una división entre cero.

Para prevenir el error que provoca la división entre cero, la declaración `if` primero verifica si `d` es igual a cero. De ser cierto, el circuito corto AND se detiene en ese punto y no realiza la división de módulo.

De esta manera, en la primera prueba **d** es igual a 2 y se realiza la operación de módulo. A continuación, el valor de **d** es igual a cero, lo que provoca que la segunda prueba falle y la operación de módulo no se ejecuta, evitando así el error que causaría la división entre cero. Finalmente, se pone a prueba el operador normal AND. Esto hace que ambos operadores sean evaluados, lo cual provoca un error en tiempo de ejecución cuando ocurre la división entre cero.

Un punto más: el circuito corto AND es también conocido como *AND condicional* y el circuito corto de OR es llamado *OR condicional*.

### Prueba esto

### Mostrar una tabla de verdad para los operadores lógicos

A continuación crearás un programa que despliega la tabla de verdad para los operadores lógicos de C#. El programa utiliza varias características estudiadas en este capítulo, incluyendo una de las secuencias de caracteres de escape de C# y los operadores lógicos. También ilustra las diferencias en la prioridad entre el operador aritmético + y los operadores lógicos.

## Paso a paso

1. Crea un nuevo archivo llamado **TablaOpLog.cs**.
2. Para asegurar que las columnas se alineen, utiliza la secuencia de escape \t con el fin de incrustar tabuladores en cada línea de salida. Por ejemplo, esta declaración **WriteLine( )** muestra el encabezado de la tabla:

```
Console.WriteLine("P\tQ\tAND\tOR\tXOR\tNOT");
```
3. Para cada una de las siguientes líneas de la tabla, utiliza tabuladores para colocar apropiadamente cada operación bajo su correspondiente encabezado.
4. A continuación presentamos el código completo del programa **TablaOpLog.cs**. Pruébalo.

```
// Presenta una tabla de verdad para los operadores lógicos.

using System;

class TablaOpLog {
    static void Main() {

        bool p, q;

        Console.WriteLine("P\tQ\tAND\tOR\tXOR\tNOT");

        p = true; q = true;
        Console.Write(p + "\t" + q + "\t");
        Console.Write((p&q) + "\t" + (p|q) + "\t");
        Console.WriteLine((p^q) + "\t" + (!p));
```

(continúa)

```

p = true; q = false;
Console.WriteLine(p + "\t" + q +"\t");
Console.WriteLine((p&q) + "\t" + (p|q) + "\t");
Console.WriteLine((p^q) + "\t" + (!p));

p = false; q = true;
Console.WriteLine(p + "\t" + q +"\t");
Console.WriteLine((p&q) + "\t" + (p|q) + "\t");
Console.WriteLine((p^q) + "\t" + (!p));

p = false; q = false;
Console.WriteLine(p + "\t" + q +"\t");
Console.WriteLine((p&q) + "\t" + (p|q) + "\t");
Console.WriteLine((p^q) + "\t" + (!p));
}
}

```

- 5.** Compila y ejecuta el programa. Aparecerá la siguiente tabla:

P	Q	AND	OR	XOR	NOT
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

- 6.** Advierte los paréntesis que rodean las operaciones lógicas dentro de las declaraciones **Write()** y **WriteLine()**. Son necesarios por la prioridad de los operadores de C#. El operador **+** tiene mayor prioridad que los operadores lógicos.
- 7.** Intenta modificar por tu cuenta este programa de manera que muestre 1 y 0 en vez de verdadero y falso.

## Pregunta al experto

**P:** Dado que los operadores de circuito corto son, en algunos casos, más eficientes que sus contrapartes normales, ¿por qué C# incluye todavía los operadores AND y OR normales?

**R:** En algunos casos será necesario evaluar ambos operandos de las operaciones AND y OR, por los efectos secundarios que producen. Analiza el siguiente ejemplo:

```

// Los efectos secundarios pueden ser importantes.

using System;

class EfectosSecundarios {
    static void Main() {
        int i;

        i = 0;

        // Aquí, i se incrementa aunque la declaración if falle.
    }
}

```

```
if(false & (++i < 100))
    Console.WriteLine("esto no será desplegado");
Console.WriteLine("la declaración if se ejecuta: " + i);
// muestra 1

// En este caso, i no se incrementa porque el operador
// de circuito corto omite el incremento.
if(false && (++i < 100))
    Console.WriteLine("esto no será desplegado");
Console.WriteLine("la declaración if se ejecuta: " + i); // sigue
siendo 1!!
}
}
```

Como lo indican los comentarios, en la primera declaración **if**, **i** se incrementa tenga éxito o no la condición. Sin embargo, cuando se utiliza el operador de circuito corto, la variable **i** no se incrementa cuando el primer operando es falso. La lección aquí es que si tu programa espera que se evalúe el operando derecho en una operación AND u OR, debes utilizar los formatos normales de esas operaciones.

## El operador de asignación

Has utilizado el operador de asignación desde el capítulo 1. Ahora es el momento de conocerlo formalmente. El *operador de asignación* es el signo de igual (**=**). Este operador funciona en C# de la misma manera como lo hace en otros lenguajes de computación. Sigue el siguiente formato general:

*variable* = *expresión*;

Donde el tipo de dato asignado a *variable* debe ser compatible con el tipo de dato de la *expresión*.

El operador de asignación tiene un atributo interesante con el que es posible que no estés familiarizado: te permite crear cadenas de asignaciones. Por ejemplo, analiza el siguiente fragmento de código:

```
int x, y, z;
x = y = z = 100; // asigna a x, y, z el valor 100
```

Este fragmento de código asigna a **x**, **y** **z** el valor 100 utilizando una sola declaración. Esto funciona porque **=** es un operador que transmite el valor asignado. De esta manera, el valor de **z = 100** es 100, el cual es asignado a **y** la cual, a su vez, lo asigna a **x**. Utilizar la “asignación en cadena” es un medio sencillo de establecer un valor común a un grupo de variables.

## Combinar asignaciones

C# proporciona operadores especiales para combinar asignaciones que simplifican la codificación de ciertas declaraciones de asignación. Comencemos con un ejemplo. La declaración de asignación:

```
x = x + 10;
```

puede escribirse utilizando una asignación combinada como la siguiente:

```
x += 10;
```

El par de operadores `+=` le indica al compilador que asigne a `x` el valor de `x` más 10.

He aquí otro ejemplo. La declaración

```
x = x - 100;
```

es lo mismo que

```
x -= 100;
```

Ambas declaraciones asignan a `x` el valor de `x` menos 100.

Existen asignaciones combinadas para muchos de los operadores binarios (aquellos que requieren dos operandos). El formato general de la combinación es:

*variable operación= expresión;*

Los operadores combinados de asignación aritméticos y lógicos aparecen en la siguiente tabla.

<code>+=</code>	<code>- =</code>	<code>* =</code>	<code>/ =</code>
<code>%=</code>	<code>&amp;=</code>	<code>  =</code>	<code>^ =</code>

Como las declaraciones de asignación combinadas son más cortas que sus equivalentes estándar, también suelen llamarse operadores de *asignaciones taquigráficas*.

Los operadores de asignación combinada proporcionan dos beneficios: primero, son más cortos que sus equivalentes “largos”; segundo, pueden traer como resultado código ejecutable más eficiente, porque el operando del lado izquierdo es evaluado sólo una vez. Por tales razones, verás que son utilizados muy a menudo en los programas C# escritos profesionalmente.

## Conversión de tipo en las asignaciones

En programación es muy común asignar el valor de un tipo a una variable de tipo diferente. Por ejemplo, tal vez quieras asignar un valor `int` a una variable `float`, como se muestra a continuación:

```
int i;  
float f;  
  
i = 10;  
f = i; // asignación de un int a un float
```

Cuando se mezclan tipos compatibles en la asignación, el valor del lado derecho de la ecuación es convertido automáticamente al tipo del lado izquierdo. De esta manera, en el fragmento anterior el valor en `i` es convertido en `float` y luego es asignado a `f`. Sin embargo, como C# revisa con rigor los tipos de datos, no todos los tipos son compatibles y, por lo mismo, no todas las conversiones de tipo son implícitamente permitidas. Por ejemplo, `bool` e `int` no son compatibles.

Cuando un tipo de dato es asignado a un tipo diferente de variable, la conversión de tipo implícita se ejecutará automáticamente siempre y cuando:

- Los dos tipos de datos sean compatibles.
- El tipo destino tenga mayor capacidad que el tipo de origen.

Cuando se cumplen estas dos condiciones se presenta una *conversión extendida*. Por ejemplo, el tipo **int** siempre tiene mayor capacidad que los valores aceptables para **byte**, y ambos son tipos de entero, por lo que se puede aplicar una conversión implícita.

Para las conversiones extendidas, los tipos numéricos, incluyendo enteros y de punto flotante, son mutuamente compatibles. Por ejemplo, el siguiente programa es perfectamente válido, porque entre **long** y **double** se realiza una conversión automática.

```
// Muestra conversión implícita de long a double.

using System;

class LongaDouble {
    static void Main() {
        long L;
        double D;

        L = 100123285L;
        D = L; ← Conversión automática
              de long a double.
        Console.WriteLine("L y D: " + L + " " + D);
    }
}
```

Aunque hay una conversión implícita de **long a double**, no sucede lo mismo a la inversa (de **double a long**), porque no se trata de una conversión extendida. Así, la siguiente versión del programa anterior es inválida:

```
// *** Este programa no se compilará. ***

using System;

class LongaDouble {
    static void Main() {
        long L;
        double D;

        D = 100123285.0;
        L = D; // Illegal!!! ← No hay conversión automática
              de double a long.
        Console.WriteLine("L y D: " + L + " " + D);

    }
}
```

Además de las restricciones ya indicadas, no existe conversión implícita entre los tipos **decimal** y **float** o **double**, ni de tipos numéricos a **char** o **bool**. Tampoco estos dos últimos son compatibles entre sí.

## Transformar tipos incompatibles

Aunque la conversión implícita de tipos es útil, no satisfará todas las necesidades de programación porque sólo se aplica a conversiones extendidas entre tipos compatibles. Para todos los demás casos de conversión debes emplear la transformación. La *transformación* es una instrucción dirigida al compilador para que convierta una expresión a un tipo especificado por el programador. Por lo mismo, requiere una conversión de tipo explícita. La transformación tiene el siguiente formato general:

*(tipo objetivo) expresión*

Donde *tipo objetivo* indica el tipo de dato al que se desea transformar la *expresión* especificada. Por ejemplo, si quieras que el tipo de dato de la expresión **x/y** sea un entero, puedes escribir

```
double x, y;  
// ...  
(int) (x / y);
```

Aquí, aunque **x** y **y** son de tipo **double**, la transformación convierte el resultado de la expresión en tipo **int**. Los paréntesis que rodean la expresión **x / y** son necesarios porque de lo contrario la transformación a **int** sólo aplicaría sobre **x** y no sobre el resultado de la división. La transformación explícita es necesaria en este caso porque no hay conversión implícita de **double** a **int**.

Cuando la transformación implica una *conversión de reducción* es posible que se pierda información. Por ejemplo, cuando se transforma un **long** a un **int**, se perderá información si el valor que contiene **long** sobrepasa el rango de capacidad de **int**, ya que los bits de alto orden serán eliminados. Cuando un valor de punto flotante se transforma en un entero, también se perderá el componente decimal debido al recorte. Por ejemplo, si el valor 1.23 es asignado a un entero, el valor resultante será 1, y el 0.23 se perderá.

El siguiente programa muestra algunos tipos de conversión que requieren transformación explícita.

```
// Muestra transformación.  
  
using System;  
  
class TransDemo {  
    static void Main() {  
        double x, y;  
        byte b;  
        int i;  
        char ch;  
  
        x = 10.0;  
        y = 3.0;  
  
        i = (int)(x / y); ← En esta transformación ocurrirá un recorte.  
        Console.WriteLine("Resultado en entero de x / y: " + i);  
    }  
}
```

```
i = 100;           No hay pérdida de información porque el
b = (byte)i;    ← tipo byte puede contener el valor 100.
Console.WriteLine("Valor de b: " + b);

i = 257;          Esta vez sí hay pérdida de información, porque
b = (byte)i;    ← un byte no puede contener el valor 257.
Console.WriteLine("Valor de b: " + b);

b = 88; // ASCII code for X
ch = (char)b;   ← La transformación es necesaria en tipos incompatibles.
Console.WriteLine("ch: " + ch);
}

}
```

Los datos de salida del programa son los siguientes:

```
Resultado en entero de x / y: 3
Valor de b: 100
Valor de b: 1
ch: X
```

En este programa, la transformación de **(x / y)** a un **int** resulta en el recorte de los componentes decimales y una parte de información se pierde. En la siguiente declaración no se pierde información cuando a **b** se le asigna el valor 100, porque **byte** tiene capacidad de contenerlo. Sin embargo, cuando se hace el intento de asignarle a **b** el valor 257 ocurre una pérdida de información porque ese valor sobrepasa el rango de **byte**. Como resultado, **b** obtiene el valor de 1 porque sólo se establece 1 bit en el orden inferior de 8 bits al que pertenece la representación binaria del valor 257. Finalmente, en la última declaración no se pierde información, pero la transformación es necesaria cuando se asigna a un tipo **byte** el valor de un tipo **char**.

## Prioridad de los operadores

La tabla 2-3 muestra el orden de prioridad de todos los operadores de C#, de mayor a menor. Esta tabla incluye varios operadores que serán analizados más tarde en este libro.

## Conversión de tipo dentro de expresiones

Es posible mezclar dos o más tipos diferentes de datos dentro de una expresión, siempre y cuando sean mutuamente compatibles. Por ejemplo, puedes mezclar **short** y **long** dentro de una expresión porque ambos son tipos numéricos. Cuando diferentes tipos de datos se mezclan en una expresión, se convierten en el mismo tipo operación por operación.

Las conversiones se realizan a través del uso de *reglas de promoción de tipo* de C#. He aquí el algoritmo definido por estas reglas para operaciones binarias:

SI un operando es **decimal**, ENTONCES el otro operando es promovido a **decimal** (a menos que sea de tipo **float** o **double**; en ese caso causará un error).

EN OTRA SITUACIÓN, SI uno de los operandos es **double**, el segundo es promovido a **double**.

### Prioridad alta

( )	[ ]	.	+(sufijo)	--(sufijo)
checked	new	sizeof	typeof	unchecked
!	~	(cast)	+(un solo operando)	-(un solo operando)
++(prefijo)	--(prefijo)			
*	/	%		
+	-			
<<	>>			
<	>	<=	>=	is
==	!=			
&				
^				
&&				
??				
? :				
=	op=	=>		

### Prioridad baja

**Tabla 2-3** La prioridad de los operadores de C#

EN OTRA SITUACIÓN, SI un operando es **float**, el segundo es promovido a **float**.

EN OTRA SITUACIÓN, SI un operando es **ulong**, el segundo es promovido a **ulong** (a menos que sea de tipo **sbyte**, **short**, **int** o **long**; en ese caso causará un error).

EN OTRA SITUACIÓN, SI un operando es **long**, el segundo es promovido a **long**.

EN OTRA SITUACIÓN, SI un operando es **uint** y el segundo es de tipo **sbyte**, **short** o **int**, ambos son promovidos a **long**.

EN OTRA SITUACIÓN, SI un operando es **uint**, el segundo será promovido a **uint**.

EN OTRA SITUACIÓN ambos operandos serán promovidos a **int**.

Hay un par de puntos importantes a destacar sobre las reglas de promoción de tipos. Primero, no todos los tipos pueden mezclarse en una expresión. Específicamente, no existe conversión implícita de **float** o **double** a **decimal**, y no es posible mezclar **ulong** con ningún tipo de entero con negativos. Para mezclar estos tipos se requiere utilizar la transformación explícita.

Segundo, pon especial atención a la última regla. Declara que si no es aplicable ninguna de las reglas anteriores, el operando será promovido al tipo **int**. De esta manera, en una expresión todos los valores **char**, **sbyte**, **byte**, **ushort** y **short** son promovidos a **int** para propósitos de cálculo. A esto se le llama *promoción a entero*. También significa que el resultado de todas las operaciones aritméticas no será más pequeño que **int**.

Es importante comprender que la promoción de tipos afecta sólo el resultado de la evaluación de la expresión donde se aplica. Por ejemplo, si el valor de una variable **byte** es promovido a **int** dentro de una expresión, fuera de ella sigue siendo **byte**. La promoción de tipos sólo afecta la evaluación de una expresión.

Sin embargo, la promoción de tipos puede conducir de alguna manera a resultados inesperados. Por ejemplo, cuando una operación aritmética implica dos valores **byte**, ocurre lo siguiente: primero, los operandos tipo **byte** son promovidos a **int**. Luego se realiza la operación que arroja un resultado tipo **int**. De esta manera, el resultado de una operación entre dos **byte** da como resultado un **int**. No es un resultado que podrías esperar intuitivamente. Analiza el siguiente programa:

```
// Una sorpresa en la promoción de tipos!
using System;
class PromDemo {
    static void Main() {
        byte b;
        int i;
        b = 10;
        i = b * b; // OK, no se requiere transformación
        b = 10;
        b = (byte)(b * b); // Se requiere transformación!!
        Console.WriteLine("i y b: " + i + " " + b);
    }
}
```

No se requiere transformación porque el resultado ya está evaluado como **int**.

Aquí se requiere transformación para asignar un **int** a un **byte**.

Contra todo pensamiento intuitivo, no se requiere transformación cuando se asigna el producto de **b \* b** a **i**, porque **b** es promovido a **int** cuando se evalúa la expresión. Así, el tipo correspondiente al resultado de **b \* b** es un **int**. Sin embargo, cuando intentas asignar el resultado de **b \* b** a **b** sí necesitas hacer una transformación ¡de regreso a **byte**! Ten en mente esto si recibes inesperadamente un mensaje de error sobre incompatibilidad de tipos, cuando todo parezca perfectamente válido.

Este mismo tipo de situación ocurre también cuando realizas operaciones sobre el tipo **char**. Por ejemplo, en el siguiente fragmento, se requiere una transformación de regreso a **char** porque la promoción de **ch1** y **ch2** a **int** dentro de la expresión:

```
char ch1 = 'a', ch2 = 'b';  
ch1 = (char) (ch1 + ch2);
```

Sin la transformación, el resultado de sumar **ch1** a **ch2** sería un tipo **int**, el cual no puede ser asignado a un **char**.

La transformación no sólo es útil para hacer conversiones de tipos dentro de una expresión. También puede utilizarse dentro de la expresión misma. Por ejemplo, analiza el siguiente programa; utiliza una transformación a **double** para obtener un componente decimal de lo que sería, de otra manera, una división entre enteros.

```
// Uso de la transformación.

using System;

class UseCast {
    static void Main() {
        int i;

        for (i = 1; i < 5; i++) {
            Console.WriteLine(i + " / 3: " + i / 3);
            Console.WriteLine(i + " / 3 con decimales: {0:#.##}",
                (double)i / 3);
            Console.WriteLine();
        }
    }
}
```

En la expresión:

```
(double) i / 3
```

la transformación a **double** provoca que la variable **i** sea convertida a tipo **double**, con lo que se asegura que permanezca el componente decimal de la división entre 3. Los datos que arroja el programa son los siguientes:

```
1 / 3: 0
1 / 3 con decimales: .33

2 / 3: 0
2 / 3 con decimales: .67

3 / 3: 1
3 / 3 con decimales: 1

4 / 3: 1
4 / 3 con decimales: 1.33
```

### Pregunta al experto

**P:** ¿Las promociones de tipo también ocurren en las operaciones con un solo operando, como en los valores negativos que llevan el signo de resta (-)?

**R:** Sí. En este tipo de operaciones, cuando el valor del operando único es menor que **int** (como **byte**, **sbyte**, **short** y **ushort**) su tipo es promovido a **int**. Lo mismo sucede con los operandos únicos de tipo **char**, también son promovidos a **int**. Más aún, si un valor **uint** es denegado, se promueve a **long**.

## Espacios en blanco y paréntesis

Una expresión en C# puede tener tabuladores y espacios en blanco para hacerla más legible. Por ejemplo, las siguientes dos expresiones realizan la misma tarea, pero la segunda es más fácil de leer:

```
x=10/y*(127-x);  
x = 10 / y * (127 - x);
```

Los paréntesis pueden utilizarse para agrupar subexpresiones, incrementando efectivamente la prioridad de las operaciones que contienen, tal y como se hace en el álgebra. El uso de paréntesis redundantes o adicionales no causará errores ni disminuirá la velocidad de ejecución del programa. Es mejor utilizar paréntesis para dejar en claro la manera como debe evaluarse cierta expresión; es conveniente tanto para el programador como para las personas que posiblemente le den mantenimiento al programa posteriormente. Por ejemplo, ¿cuál de las dos expresiones siguientes es más fácil de leer?

```
x = y/3-34*temp+127;  
x = (y/3) - (34*temp) + 127;
```

### Prueba esto

## Calcular los pagos regulares de un préstamo

Como se mencionó con anterioridad, el tipo **decimal** es especialmente útil para cálculos de dinero. El siguiente programa muestra esta capacidad; calcula los pagos regulares de un préstamo, como puede ser el crédito para un automóvil. Dados el monto principal, la vida del crédito, la cantidad de pagos por año y la tasa de interés, el programa calculará los pagos. Como se trata de un cálculo financiero, tiene sentido utilizar el tipo **decimal** para representar los datos. En el programa también se utiliza la transformación de tipos y otros métodos de la biblioteca de C#.

Para calcular los pagos debes utilizar la siguiente fórmula:

$$\text{Pago} = \frac{\text{TasaInt} * (\text{Monto} / \text{PagoAnual})}{1 - ((\text{TasaInt} / \text{PagoAnual}) + 1)^{-\text{PagoAnual} * \text{NumAños}}}$$

donde TasaInt especifica la tasa de interés, Monto es la cantidad total del préstamo, PagoAnual es la cantidad de pagos que se realizan por año y NumAños es la cantidad de años durante los cuales se pagará el crédito.

Advierte que en la fórmula se indica que debes aumentar un valor a la potencia de otro. Para hacerlo, utilizarás el método matemático de C# **Math.Pow()**. He aquí cómo debes invocarlo:

```
resultado = Math.Pow(base, expresión);
```

**Pow()** regresa el valor de *base* aumentado a la potencia de *expresión*. Los argumentos de **Pow()** deben ser de tipo **double**, y regresa el valor resultante en el mismo tipo. Esto significa que necesitas transformarlo para convertir el **double** en **decimal**.

(continúa)

## Paso a paso

1. Crea un nuevo archivo llamado **PagoReg.cs**.
2. He aquí las variables que serán utilizadas en el programa:

```
decimal Monto;           // monto original
decimal TasaInt;         // tasa de interés como decimal, como 0.075
decimal PagoAnual;        // cantidad de pagos por año
decimal NumAños;          // número de años
decimal Payment;          // el pago regular
decimal numer, denom;    // variables de trabajo temporales
double b, e;              // base y exponente para invocar Pow()
```

Como la mayoría de los cálculos se realizarán utilizando el tipo de dato **decimal**, la mayoría de las variables corresponden a ese tipo.

Advierte que cada declaración de variable va acompañada de un comentario que explica su uso. Esto ayuda a que cualquier otra persona que lea tu programa conozca el propósito de cada variable. Aunque no incluiremos tales comentarios detallados en los programas cortos que aparecen en el presente libro, se trata de una buena práctica que debes seguir sobre todo cuando tus programas se hagan más grandes y complejos.

3. Añade las siguientes líneas de código, que especifican la información sobre el préstamo. En este caso, el monto es \$10 000, la tasa de interés es 7.5 por ciento, la cantidad de pagos por año son 12 y la vida del crédito es de 5 años.

```
Monto = 10000.00m;
TasaInt = 0.075m;
PagoAnual = 12.0m;
NumAños = 5.0m;
```

4. Añade las líneas que realizan el cálculo financiero:

```
numer = TasaInt * Monto / PagoAnual;
e = (double) -(PagoAnual * NumAños);
b = (double)(TasaInt / PagoAnual) + 1;
denom = 1 - (decimal) Math.Pow(b, e);
Payment = numer / denom;
```

Observa con atención cómo debe utilizarse la transformación para transmitir valores a **Pow()** y para convertir el valor de respuesta. Recuerda que no hay conversiones implícitas entre **decimal** y **double** en C#.

5. Finaliza el programa presentando los pagos regulares, como se muestra a continuación:

```
Console.WriteLine("El pago es de {0:C}", Payment);
```

6. He aquí el programa **PagoReg.cs**. completo:

```
// Calcula los pagos regulares de un préstamo.
```

```
using System;

class PagoReg {
    static void Main() {
        decimal Monto;           // monto original
        decimal TasaInt;         // tasa de interés como decimal, como 0.075
        decimal PagoAnual;       // cantidad de pagos por año
        decimal NumAños;         // número de años
        decimal Payment;         // el pago regular
        decimal numer, denom;   // variables de trabajo temporales
        double b, e;             // base y exponente para invocar Pow()

        Monto = 10000.00m;
        TasaInt = 0.075m;
        PagoAnual = 12.0m;
        NumAños = 5.0m;

        numer = TasaInt * Monto / PagoAnual;

        e = (double) -(PagoAnual * NumAños);
        b = (double)(TasaInt / PagoAnual) + 1;

        denom = 1 - (decimal) Math.Pow(b, e);

        Payment = numer / denom;

        Console.WriteLine("El pago es de {0:C}", Payment);
    }
}
```

El resultado que arroja el programa:

El pago es de \$200.38

Antes de seguir adelante, tal vez quieras hacer que el programa calcule pagos regulares para diferentes montos, períodos y tasas de interés.



## Autoexamen Capítulo 2

1. ¿Por qué C# especifica estrictamente el rango y comportamiento de sus tipos primitivos?
2. ¿Qué es el tipo carácter de C# y en qué se diferencia de sus equivalentes en otros lenguajes de programación?
3. Un valor **bool** puede tener cualquier valor que quieras porque cualquier valor diferente a cero es verdadero. ¿Cíerto o falso?

- 4.** Considerando los siguientes datos de salida:

Uno  
Dos  
Tres

Utiliza una sola cadena de caracteres y secuencias de escape para armar la declaración **WriteLine()** que los genera.

- 5.** ¿Qué error hay en el siguiente fragmento de código?

```
for(i = 0; i < 10; i++) {  
    int suma;  
  
    suma = suma + 1;  
}  
Console.WriteLine("Suma es: " + suma);
```

- 6.** Explica la diferencia entre los formatos prefijo y sufijo en el operador de incremento.
- 7.** Muestra cómo debe utilizarse un circuito corto de AND para evitar un error de división entre cero.
- 8.** En una expresión, ¿a qué tipo son promovidos **byte** y **short**?
- 9.** ¿Cuál de los siguientes tipos no puede ser mezclado en una expresión con el tipo **decimal**?
- A.** float
  - B.** int
  - C.** uint
  - D.** byte
- 10.** En términos generales, ¿cuándo se requiere una transformación?
- 11.** Escribe un programa que presente todos los números primos entre 2 y 100.
- 12.** Reescribe el programa de la tabla de verdad que aparece en *Prueba esto: Mostrar una tabla de verdad para los operadores lógicos*, de tal manera que utilice cadenas de caracteres verbatim con tabuladores y saltos de línea insertados en lugar de secuencias de escape.

# Capítulo 3

Declaraciones para el  
control del programa

## Habilidades y conceptos clave

- Ingresar caracteres desde el teclado
  - **if** y **for**
  - **switch**
  - **while**
  - **do-while**
  - **break**
  - **continue**
  - **goto**
- 

Este capítulo está dedicado a explicar las declaraciones que controlan el flujo de ejecución del programa, las cuales se agrupan en tres grandes categorías: *selección*, que incluyen **if** y **switch**; *reiteración*, que incluyen los loops **for**, **while**, **do-while** y **foreach**, y *trayectoria*, que incluyen **break**, **continue**, **goto**, **return** y **throw**. Con excepción de **return**, **foreach** y **throw**, que se analizan más tarde en este mismo libro, las restantes son examinadas con detalle en este capítulo.

## Ingresar caracteres desde el teclado

Antes de examinar las declaraciones de control de C#, haremos un pequeño paréntesis que te permitirá comenzar a trabajar con programas interactivos. Hasta este momento, los programas ejemplo de este libro han mostrado información *hacia* el usuario, pero no han recibido información *desde* el usuario. Has utilizado las características de salida de la consola, pero no has ocupado sus características de entrada (proporcionadas por el teclado). Ahora comenzarás a utilizar datos de entrada leyendo los caracteres que son escritos con el teclado de la computadora.

Para leer un carácter proveniente del teclado, invoca **Console.Read()**. Este método espera hasta que el usuario presione una tecla para luego mostrarla en la consola. El carácter es regresado como un entero, por lo que debes transformarlo en **char** para que sea válido asignarlo a una variable de ese tipo. Por omisión, los datos de entrada de la consola son almacenados temporalmente en la memoria, por lo que debes oprimir la tecla ENTER para que los caracteres que escribas sean enviados al programa. A continuación presentamos un programa que lee un carácter proveniente del teclado:

```
// Lee un carácter proveniente del teclado.  
  
using System;  
  
class EntraTecla {  
    static void Main() {  
        char ch;
```

```
Console.WriteLine("Presiona cualquier tecla seguida de ENTER: ");

// Lee la tecla proveniente del teclado.
ch = (char) Console.Read(); ← Lee un carácter del teclado.

Console.WriteLine("Tu tecla es: " + ch);
}

}
```

Un ejemplo de los datos de salida:

```
Presiona cualquier tecla seguida de ENTER: t
Tu tecla es: t
```

El hecho de que **Read()** almacene temporalmente los datos en la memoria suele ser fastidioso. Cuando oprimes la tecla ENTER, un retorno de carro, la secuencia de caracteres escrita es ingresaada al flujo entrada. Más aún, estos caracteres dependen de la memoria de entrada hasta que son leídos por la aplicación que se ejecuta. De esta manera, para algunas aplicaciones será necesario que los elimines de la memoria temporal (incorporándolos a la aplicación) antes de ejecutar la siguiente operación de entrada. Verás un ejemplo de esto más tarde en este mismo capítulo.

## La declaración if

En el capítulo 1 vimos una introducción a la declaración **if**; ahora la estudiaremos con detalle. El formato completo de la declaración **if** es el siguiente:

```
if(condición) declaración;
else declaración;
```

donde los objetivos de **if** y **else** son declaraciones sencillas. La cláusula **else** es opcional. Como se puede utilizar un bloque de código en cualquier punto donde una declaración sencilla sea válida, los objetivos de **if** y **else** también pueden ser bloques de código. En tales casos, el formato general de **if** es el siguiente:

```
if(condición)
{
    secuencia de declaraciones
}
else
{
    secuencia de declaraciones
}
```

Cuando la expresión condicional sea verdadera, se ejecutará el bloque objetivo de **if**; en caso contrario se ejecutará el de **else**, si existe. En ninguna circunstancia se ejecutarán ambos. La expresión condicional que controla la declaración **if** debe generar un valor **bool**.

Para mostrar el uso de **if**, escribiremos un sencillo juego computarizado de adivinanza, apropiado para niños pequeños. En la primera versión del juego, el programa tiene grabada una letra entre la A y la Z y pedirá al usuario que la adivine. Si el usuario oprime la tecla correcta, el programa mostrará un mensaje que diga **\*\* Correcto \*\***. He aquí el código:

```
// Juego de adivinar una letra.

using System;

class Adivina {
    static void Main() {
        char ch, respuesta = 'K';

        Console.WriteLine("Estoy pensando en una letra entre A y Z.");
        Console.Write("¿Puedes adivinarla?: ");

        ch = (char) Console.Read(); // obtiene la opción del usuario

        if (ch == respuesta) Console.WriteLine("** Correcto **");
    }
}
```

El programa espera la respuesta del usuario y luego lee el carácter escrito con el teclado. Utilizando una declaración **if**, compara entonces el carácter escrito con la respuesta correcta, en este caso la letra K. Si el usuario oprimió la tecla correspondiente a la letra K, el programa mostrará el mensaje predeterminado. Cuando pruebes el programa, recuerda que debes ingresar la K mayúscula.

Llevemos el juego un paso adelante. La siguiente versión utiliza la cláusula **else** para presentar un mensaje cuando se selecciona una letra equivocada.

```
// Juego de adivinar una letra, 2a. versión.

using System;

class Adivina2 {
    static void Main() {
        char ch, respuesta = 'K';

        Console.WriteLine("Estoy pensando en una letra entre A y Z.");
        Console.Write("¿Puedes adivinarla?: ");

        ch = (char) Console.Read(); // obtiene la opción del usuario

        if(ch == respuesta) Console.WriteLine("** Correcto **");
        else Console.WriteLine("...Lo siento, estás equivocado.");
    }
}
```

## if anidados

Un **if anidado** es una declaración **if** cuyo objetivo es otro **if** o un **else**. Los **if** anidados son muy comunes en la programación. Lo más importante que debes recordar sobre la anidación de **if** en C# es que la cláusula **else** siempre está asociada con la declaración **if** más cercana que se encuentra dentro de su mismo bloque, y que no esté previamente asociado con otro **else**. He aquí un ejemplo:

```
if(i == 10) { ← Este if corresponde a este else.  
    if(j < 20) a = b;  
    if(k > 100) c = d; ← Este if corresponde a este else.  
    else a = c; // este else corresponde a if(k > 100) ←  
}  
else a = d; // este else corresponde a if(i == 10) ←
```

Como lo indican los comentarios, la cláusula **else** final no está asociada con **if(j < 20)**, porque no se localiza en el mismo bloque (a pesar de ser el **if** más cercano sin un **else** asociado). En lugar de eso, la cláusula **else** final está asociada con **if(i == 10)**. Por tanto, el **else** interno está asociado con **if(k > 100)**, porque es el más cercano dentro del mismo bloque.

Puedes utilizar **if** anidados para mejorar el juego de adivinanza. El complemento presentará al usuario un mensaje de retroalimentación en caso de que la respuesta sea errónea.

```
// Juego de adivinar una letra, 3a. versión.  
  
using System;  
  
class Adivina3 {  
    static void Main() {  
        char ch, respuesta = 'K';  
  
        Console.WriteLine("Estoy pensando en una letra entre A y Z.");  
        Console.Write("¿Puedes adivinarla?: ");  
  
        ch = (char) Console.Read(); // obtiene la opción del usuario  
  
        if(ch == respuesta) Console.WriteLine("** Correcto **");  
        else {  
            Console.Write("...Lo siento, estás ");  
            // Un if anidado ↓  
            if(ch < respuesta) Console.WriteLine("muy bajo");  
            else Console.WriteLine("muy alto");  
        }  
    }  
}
```

Un ejemplo de los datos de salida:

```
Estoy pensando en una letra entre A y Z.  
¿Puedes adivinarla?: Z  
...Lo siento, estás muy alto
```

## La escalera if-else-if

Un constructor muy utilizado en programación, basado en la anidación de **if**, es la *escalera if-else-if*. Tiene el siguiente formato:

```
if(condición)
    declaración;
else if(condición)
    declaración;
else if(condición)
    declaración;

.
.
.

else
    declaración;
```

Las expresiones condicionales son evaluadas de arriba hacia abajo. En cuanto se descubre una condición verdadera, la declaración asociada a ella se ejecuta, y el resto de la escalera se omite. En caso de que ninguna de las condiciones sea verdadera, se ejecuta la última cláusula **else**. Por lo regular esta última cláusula actúa como la condición por omisión; es decir, sólo se ejecuta en caso de que las demás sean falsas. En caso de que no se declare la última cláusula **else** y el resto de las condiciones sean falsas, no se ejecuta ninguna acción.

El siguiente programa muestra la escalera **if-else-if**:

```
// Muestra la escalera if-else-if.

using System;

class Escalera {
    static void Main(){
        int x;

        for(x=0; x<6; x++){
            if(x==1)
                Console.WriteLine("x es uno");
            else if(x==2)
                Console.WriteLine("x es dos");
            else if(x==3)
                Console.WriteLine("x es tres");
            else if(x==4)
                Console.WriteLine("x es cuatro");
            else
                Console.WriteLine("x no está entre 1 y 4");
        }
    }
}
```

Ésta es la declaración por omisión.

El programa genera los siguientes datos de salida:

```
x no está entre 1 y 4
x es uno
x es dos
x es tres
x es cuatro
x no está entre 1 y 4
```

Como puedes ver, la cláusula **else** final sólo se ejecuta si ninguna de las declaraciones anteriores es verdadera.

## La declaración **switch**

La segunda declaración de selección de C# es **switch**, que proporciona alternativas múltiples, es decir, permite que el programa haga una selección entre diversas opciones. Aunque una serie de declaraciones **if** anidadas pueden realizar pruebas múltiples, para muchas situaciones **switch** es una solución más eficiente. Funciona de la siguiente manera: el valor de una expresión es comparado sucesivamente contra una lista de constantes. Cuando se encuentra una coincidencia, la declaración asociada con ella se ejecuta. El formato general de la declaración **switch** es la siguiente:

```
switch(expresión) {
    case constante1:
        secuencia de declaraciones
        break;
    case constante2:
        secuencia de declaraciones
        break;
    case constante3:
        secuencia de declaraciones
        break;
    .
    .
    .
    default:
        secuencia de declaraciones
        break;
}
```

La expresión **switch** debe ser un tipo integral como **char**, **byte**, **short** o **int**; un tipo de numeración, o un tipo **string**. (Los tipos de numeración y **string** se describen más adelante en el libro.) De manera que los valores de punto flotante, por ejemplo, no son válidos. Con frecuencia, la expresión que controla **switch** es sencillamente una variable. Las constantes **case** deben ser de tipo compatible con la expresión. No es válido tener dos constantes **case** con valor idéntico dentro de la misma expresión **switch**.

La secuencia **default** es ejecutada en caso de que ninguna constante **case** coincida con la expresión. **Default** es opcional; si no se incluye, el programa no ejecuta ninguna acción en caso de no encontrar una coincidencia. Por otra parte, cuando se encuentra una coincidencia, la declaración asociada con ese **case** es ejecutada, todo el código hasta que el programa encuentra la instrucción **break**.

El siguiente programa muestra el funcionamiento de **switch**:

```
// Muestra el funcionamiento de switch.

using System;

class SwitchDemo {
    static void Main() {
        int i;

        for(i=0; i < 10; i++)
            switch (i) { ← El valor de i determina cuál
                case 0:           declaración case se ejecuta.
                    Console.WriteLine("i es cero");
                    break;
                case 1:
                    Console.WriteLine("i es uno");
                    break;
                case 2:
                    Console.WriteLine("i es dos");
                    break;
                case 3:
                    Console.WriteLine("i es tres");
                    break;
                case 4:
                    Console.WriteLine("i es cuatro");
                    break;
                default:
                    Console.WriteLine("i es cinco o más");
                    break;
    }
}
```

Los datos de salida generados por el programa son los siguientes:

```
i es cero
i es uno
i es dos
i es tres
i es cuatro
i es cinco o más
```

Como puedes ver, a lo largo del loop las declaraciones asociadas con las constantes **case** que coinciden con el valor de **i** son ejecutadas. Las demás son omitidas. Cuando el valor de **i** es cinco o superior, ninguna constante **case** coincide, así que se ejecuta la declaración asociada con **default**.

En el ejemplo anterior, la declaración **switch** está controlada por una variable **int**. Como ya se explicó, puedes controlar **switch** con cualquier tipo integral, incluido **char**. A continuación presentamos un ejemplo que utiliza una expresión **char** y constantes **case** del mismo tipo.

```
// Usa un tipo char para controlar switch.

using System;

class SwitchDemo2 {
    static void Main() {
        char ch;

        for(ch='A'; ch <= 'E'; ch++)
            switch(ch) {
                case 'A':
                    Console.WriteLine("ch es A");
                    break;
                case 'B':
                    Console.WriteLine("ch es B");
                    break;
                case 'C':
                    Console.WriteLine("ch es C");
                    break;
                case 'D':
                    Console.WriteLine("ch es D");
                    break;
                case 'E':
                    Console.WriteLine("ch es E");
                    break;
            }
    }
}
```

Los datos de salida generados por el programa son:

```
ch es A
ch es B
ch es C
ch es D
ch es E
```

Advierte que este ejemplo no incluye el caso **default**, recuerda que es opcional. Cuando no se necesita se puede dejar fuera.

En C# es un error que la secuencia de declaraciones asociada con un **case** continúe en el siguiente *caso*. A esto se le llama *regla de no intromisión*; por eso las secuencias **case** terminan con una instrucción **break**. (Puedes omitir la intromisión de otras maneras, pero **break** es por

mucho la más utilizada.) Cuando se encuentra dentro de una secuencia de declaración de un **case**, **break** hace que el flujo del programa salga totalmente de la declaración **switch** y se reanude en la siguiente declaración. Un punto más: la secuencia **default** tampoco debe entrometerse y por esa razón también finaliza con una instrucción **break**.

Aunque no puedes permitir que una secuencia **case** se entrometa con otra, sí es válido tener dos o más etiquetas **case** para la misma secuencia de código, como se muestra en el siguiente ejemplo:

```
switch(i) {  
    case 1:  
    case 2: ← Estos casos hacen referencia a la misma secuencia de código.  
    case 3: Console.WriteLine("i es 1, 2 o 3");  
        break;  
    case 4: Console.WriteLine("i es 4");  
        break;  
}
```

En este fragmento, si **i** tiene el valor 1, 2 o 3, se ejecuta la primera declaración **WriteLine()**. Si el valor es 4, se ejecuta la segunda declaración **WriteLine()**. *Aglomerar* diferentes **case** respeta la regla de no intromisión porque todos los **case** utilizan la misma secuencia de declaraciones.

Aglomerar etiquetas **case** es una técnica comúnmente empleada cuando varios **case** comparan código. Por ejemplo, es utilizada para categorizar letras minúsculas del alfabeto en vocales y consonantes:

```
// Categoriza letras minúsculas en vocales y consonantes.  
  
using System;  
  
class VocalesYConsonantes {  
    static void Main() {  
        char ch;  
  
        Console.Write("Escribe una letra: ");  
        ch = (char) Console.Read();  
        switch(ch) {  
            case 'a':  
            case 'e':  
            case 'i':  
            case 'o':  
            case 'u':  
            case 'y':  
                Console.WriteLine("La letra es una vocal.");  
                break;  
            default:  
                Console.WriteLine("La letra es una consonante.");  
                break;  
        }  
    }  
}
```

## Pregunta al experto

**P:** ¿Bajo qué circunstancias debo utilizar la escalera if-else-if en lugar de switch cuando necesite codificar opciones múltiples?

**R:** En general, utiliza la escalera **if-else-if** cuando las condiciones del proceso de selección no dependan de un solo valor. Por ejemplo, analiza la siguiente secuencia **if-else-if**:

```
if(x == 10) // ...
else if(ch == 'a') // ...
else if(hecho == true) // ...
```

Esta secuencia no puede trasladarse a una declaración **switch** porque las tres condiciones implican variables diferentes (y tipos de datos diferentes). ¿Qué variable controlaría el **switch**? De la misma manera, necesitarás utilizar una escalera **if-else-if** cuando trabajes con valores de punto flotante u otros objetos cuyo tipo sea incompatible con los utilizados por las expresiones **switch**. Por último, las declaraciones **switch** sólo pueden probar igualdades. Si estás buscando relaciones diferentes, como menor que o diferente a, debes utilizar la escalera **if-else-if**. Por ejemplo:

```
if(x < 10) // ...
else if(y >= 0) // ...
else if(z != -1) // ...
```

Esta secuencia tampoco puede representarse con una declaración **switch**.

Si este ejemplo fuera escrito sin una aglomeración de **case**, la misma declaración **WriteLine()** tendría que duplicarse seis veces. La aglomeración de **case** evita esta duplicación redundante.

## Declaraciones switch anidadas

Es posible tener un **switch** como parte de una secuencia de declaraciones de un **switch** externo. A esto se le llama **switch anidado**. Las constantes de los **switch** interno y externo pueden contener valores comunes, sin que esto cause conflictos. Por ejemplo, el siguiente fragmento de código es perfectamente aceptable:

```
switch(ch1) {
    case 'A': Console.WriteLine("Esta A es parte del switch externo.");
    switch(ch2) { ← Un switch anidado.
        case 'A':
            Console.WriteLine("Esta A es parte del switch interno");
            break;
        case 'B': // ...
    } // fin del switch interno
    break;
    case 'B': // ...
```

## Pregunta al experto

**P:** En C, C++ y Java un **case** puede continuar (es decir, entrometerse) en el siguiente **case**. ¿Por qué no se permite hacer lo mismo en C#?

**R:** Existen dos razones por las que C# instituyó la regla de no intromisión para **case**. Primero, permite disponer en un nuevo orden las declaraciones **case**. Ese nuevo orden no sería posible si un **case** se entrometiera en otro. Segundo, hacer obligatorio que cada **case** finalice explícitamente, evita que el programador permita la intromisión de un **case** en otro por error.

## Prueba esto

### Comenzar a construir un sistema de ayuda C#

Ahora comenzarás a construir un sistema de ayuda sencillo que muestra la sintaxis de las declaraciones de control de C#. Este sistema de ayuda será enriquecido a lo largo del presente capítulo. Esta primera versión muestra un menú que contiene las declaraciones de control y luego espera a que selecciones una. Después de la elección se muestra la sintaxis de esa declaración. En la primera versión del programa, la ayuda sólo estará disponible para las declaraciones **if** y **switch**. Las demás declaraciones se añaden en las siguientes secciones *Prueba esto*.

## Paso a paso

1. Crea un archivo llamado **Ayuda.cs**.
2. El programa comienza desplegando el siguiente menú:

```
Ayuda sobre:  
1. if  
2. switch  
Seleccione una:
```

Para realizar esto, utilizarás la secuencia de declaraciones que se muestra a continuación:

```
Console.WriteLine("Ayuda sobre: ");
Console.WriteLine(" 1. if");
Console.WriteLine(" 2. switch");
Console.Write("Seleccione una: ");
```

3. El programa obtiene la selección del usuario invocando el método **Console.Read()**, como se muestra aquí:

```
choice = (char) Console.Read();
```

4. Una vez que se ha obtenido la solución, el programa utiliza la declaración **switch**, que se muestra a continuación, para mostrar la sintaxis de la declaración seleccionada:

```
switch(choice) {  
    case '1':  
        Console.WriteLine("El if:\n");  
        Console.WriteLine("if(condición) declaración;");  
        Console.WriteLine("declaración else;");  
        break;  
    case '2':  
        Console.WriteLine("El switch:\n");  
        Console.WriteLine("switch(expresión) {");  
        Console.WriteLine("  constante case:");  
        Console.WriteLine("    secuencia de declaraciones");  
        Console.WriteLine("    break;");  
        Console.WriteLine("  // ...");  
        Console.WriteLine("}");  
        break;  
    default:  
        Console.Write("No se encontró la selección.");  
        break;  
}
```

Advierte que la cláusula **default** atrapa las opciones inválidas. Por ejemplo, si el usuario escribe el número 3, ninguna constante **case** coincidirá y la secuencia **default** será ejecutada.

5. Aquí está el programa **Ayuda.cs** completo:

```
// Un sistema de ayuda sencillo.  
  
using System;  
  
class Ayuda {  
    static void Main() {  
        char choice;  
  
        Console.WriteLine("Ayuda sobre:");  
        Console.WriteLine("  1. if");  
        Console.WriteLine("  2. switch");  
        Console.Write("Seleccione una: ");  
        choice = (char) Console.Read();  
  
        Console.WriteLine("\n");  
  
        switch (choice) {  
            case '1':  
                Console.WriteLine("El if:\n");  
                Console.WriteLine("if(condición) declaración;");  
                Console.WriteLine("declaración else;");  
                break;  
            case '2':  
                Console.WriteLine("El switch:\n");  
                Console.WriteLine("switch(expresión) {");  
                Console.WriteLine("  constante case:");  
        }
```

(continúa)

```
        Console.WriteLine("    secuencia de declaraciones");
        Console.WriteLine("        break;");
        Console.WriteLine("    // ...");
        Console.WriteLine("}");
        break;
    default:
        Console.Write("No se encontró la selección.");
        break;
    }
}
```

He aquí un ejemplo de los datos de salida:

```
Ayuda sobre:
1. if
2. switch
Elija uno: 1

El if:

if(condición) declaración;
declaración else;
```

---

## El loop **for**

Desde el capítulo 1 has utilizado formatos sencillos del loop **for**. Tal vez te sorprendas cuando descubras lo poderoso y flexible que es. Comenzaremos por revisar los aspectos básicos, iniciando por los formatos más tradicionales de **for**.

El formato general de **for** para repetir una sola declaración es el siguiente:

```
for(inicialización; condición; reiteración) declaración;
```

El formato general para repetir un bloque es:

```
for(inicialización; condición; reiteración)
{
    secuencia de declaraciones
}
```

La *inicialización* es por lo regular una declaración de asignación que establece el valor inicial de la *variable de control del loop*, la cual funciona como el contador que controla las reiteraciones. La *condición* es una expresión booleana que determina hasta cuándo se repetirá el loop. La expresión *reiteración* determina la magnitud del cambio en la variable de control cada vez que se repite el loop. Advierte que estas tres secciones principales del loop deben ir separadas por punto y coma. El loop **for** se ejecutará y continuará ejecutándose mientras la condición de control sea verdadera. Una vez que esta condición genere un valor falso, el loop concluirá y el flujo del programa avanzará a la siguiente declaración inmediata posterior a **for**.

El siguiente programa utiliza un loop **for** para presentar las raíces cuadradas de los números entre el 1 y el 99. También muestra los errores de redondeo para cada raíz cuadrada.

```
// Presenta las raíces cuadradas del 1 al 99 y los errores de redondeo.

using System;

class RaízCuadrada {
    static void Main() {
        double num, raíz, rerr;

        for(num = 1.0; num < 100.0; num++) {
            raíz = Math.Sqrt(num);
            Console.WriteLine("Raíz cuadrada de " + num +
                " es " + raíz);

            // Calcula error de redondeo.
            rerr = num - (raíz * raíz);
            Console.WriteLine("El error de redondeo es " + rerr);
            Console.WriteLine();
        }
    }
}
```

Advierte que el error de redondeo se calcula primero obteniendo el cuadrado de cada número, el resultado luego es restado del número original, lo cual arroja el error de redondeo. Por supuesto, en algunos casos los errores de redondeo ocurren cuando se obtienen los cuadrados, por lo que en ocasiones ¡el error en sí es redondeado! Este ejemplo ilustra el hecho de que los cálculos con valores de punto flotante no siempre son tan precisos como suponemos que deben ser.

El loop **for** puede aplicarse a cantidades positivas o negativas, y puede cambiar la magnitud de la variable de control en cualquier cantidad. Por ejemplo, el siguiente loop presenta los números existentes entre el 100 y el -100, con un decremento de cinco unidades:

```
// Una sucesión negativa del loop for.
for(x = 100; x > -100; x -= 5)
    Console.WriteLine(x);
}
```

Un punto importante sobre los loops **for** es que la expresión condicional siempre es probada al inicio de la reiteración. Esto significa que el código escrito dentro del loop no se ejecutará en absoluto si la condición es falsa desde el principio. He aquí un ejemplo:

```
for(cuenta=10; cuenta < 5; cuenta++)
    x += cuenta; // esta declaración no se ejecutará
```

Este loop nunca se ejecutará porque la variable de control, **cuenta**, es mayor que cinco cuando el loop inicia. Esto hace que la expresión condicional, **cuenta<5**, sea falsa desde el principio; por lo mismo, no ocurrirá ni una sola repetición del loop.

## Algunas variantes del loop for

**For** es una de las declaraciones más versátiles de C# porque permite una gran cantidad de variaciones. Por ejemplo, se pueden utilizar múltiples variables de control. Observa con cuidado el siguiente programa:

```
// Uso de comas en la declaración for.

using System;

class Coma {
    static void Main() {
        int i, j;

        for(i=0, j=10; i < j; i++, j--) ← Nota las dos variables
            Console.WriteLine("i y j: " + i + " " + j);
    }
}
```

El resultado del programa es el siguiente:

```
i y j: 0 10
i y j: 1 9
i y j: 2 8
i y j: 3 7
i y j: 4 6
```

Aquí las dos declaraciones de inicialización son separadas por comas, al igual que las expresiones de reiteración. Cuando inicia el loop, se inicializan **i** y **j**. Cada vez que se repite el loop, **i** se incrementa y **j** se reduce. Múltiples variables de control en el loop por lo regular son convenientes para simplificar ciertos algoritmos. Es posible insertar cualquier cantidad de inicializadores y reiteradores, pero en la práctica, incluir más de dos hace muy difícil el manejo de **for**.

La condición de control del loop puede ser cualquier expresión válida que genere un resultado tipo **bool**. Aunque poco conveniente, es posible dejar fuera la variable de control. En el siguiente ejemplo, el loop continúa ejecutándose hasta que el usuario escribe una **S** con el teclado.

```
// Loop continuo hasta que el usuario escribe una S.

using System;

class ForTest {
    static void Main() {
        int i;

        Console.WriteLine("Presione S para detener.");
        for(i = 0; (char)Console.Read() != 'S'; i++)
            Console.WriteLine("Repetición #" + i);
    }
}
```

## Piezas perdidas

Algunas variaciones interesantes del loop **for** se crean cuando se deja en blanco alguna pieza fundamental del loop. En C# es posible dejar en blanco cualquiera de las porciones del **for**: la inicialización, la condición o la reiteración. Por ejemplo, observa el siguiente programa:

```
// Partes de for pueden ser dejadas en blanco.

using System;

class Vacío {
    static void Main() {
        int i;

        for(i = 0; i < 10; ) { ← La expresión de reiteración no existe.
            Console.WriteLine("Repetición #" + i);
            i++; // variable de control de incremento del loop
        }
    }
}
```

Aquí, la expresión de reiteración del **for** está vacía. La variable de control del loop **i** se incrementa dentro del cuerpo del loop. Esto significa que cada vez que se ejecute la repetición, **i** se consulta para comprobar si su valor es igual a 10, pero no se realiza ninguna acción adicional. Por supuesto, como **i** se incrementa dentro del cuerpo del loop, éste se ejecuta normalmente, mostrando los siguientes datos:

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

En el siguiente ejemplo, la parte de inicialización también es extraída de **for**.

```
// Mueve más partes del loop for.

using System;

class Vacío2 {
    static void Main() {
        int i;

        i = 0; // mueve la inicialización fuera del loop ← La expresión de inicialización
        for ( ; i < 10; ) { ← La expresión de inicialización
            se encuentra fuera del loop.
    }
}
```

```
        Console.WriteLine("Repetición #" + i);
        i++; // variable de control de incremento del loop
    }
}
```

En esta versión, **i** se inicializa antes de que el loop comience, en lugar de hacerlo como parte de **for**. Normalmente, querrás inicializar la variable de control del loop dentro de **for**. Colocar la parte de inicialización fuera del loop por lo general se hace sólo cuando el valor inicial es producto de un proceso complejo, cuya naturaleza no permite que se localice dentro de la declaración **for**.

## El loop infinito

Puedes crear un *loop infinito* (un loop que nunca concluye) con el uso de **for**, dejando en blanco su expresión condicional. Por ejemplo, el siguiente fragmento muestra la manera en que muchos programadores de C# crean loops infinitos:

```
for(;;) // loop infinito intencional
{
    //...
}
```

Este loop se ejecutará por siempre. Aunque existen ciertas tareas de programación que requieren una repetición infinita, como los procesos de instrucción de los sistemas operativos, la mayoría de los “loops infinitos” son en realidad repeticiones con requerimientos especiales de terminación. Cerca del final de este capítulo aprenderás a detener un loop de este tipo. (Pista: se hace utilizando la declaración **break**.)

## Loops sin cuerpo

En C#, el cuerpo asociado con el loop **for** (o cualquier otro loop) puede estar vacío. Esto se debe a que la sintaxis de una declaración en blanco es perfectamente válida. Los loops sin cuerpo suelen ser útiles, por ejemplo: el siguiente programa utiliza esta táctica para sumar los números del 1 al 5:

```
// El cuerpo del loop puede estar vacío.

using System;

class Vacío3 {
    static void Main() {
        int i;
        int suma = 0;

        // Suma los números del 1 al 5.
        for(i = 1; i <= 5; suma += i++) ; ←———— ¡Este loop no tiene cuerpo!

        Console.WriteLine("La suma es " + suma);
    }
}
```

El resultado que genera el programa es el siguiente:

```
La suma es 15
```

Observa que el proceso de adición es controlado completamente dentro de la declaración **for** y no es necesario ningún cuerpo. Pon especial atención a la reiteración:

```
suma += i++
```

Que no te intimiden este tipo de declaraciones. Son muy comunes en los programas C# escritos profesionalmente y son fáciles de entender si las divides en sus partes constitutivas. En prosa, esta declaración diría: “añade a **suma** el valor de **suma** más **i**, luego incrementa **i**”. De esta manera, sería la misma declaración que:

```
suma = suma + 1;  
i++;
```

## Declarar variables de control de reiteración dentro del loop for

Muy seguido, la variable que controla el loop **for** sólo es necesaria para ese propósito y no se requiere en otra parte del programa. Cuando suceda así, es posible declarar la variable dentro de la porción de inicialización de **for**. Por ejemplo, el siguiente programa calcula tanto la sumatoria como el factorial de los números del 1 al 5, y declara la variable de control **i** dentro de **for**:

```
// Declara la variable de control del loop dentro de for.  
  
using System;  
  
class ForVar {  
    static void Main() {  
        int sum = 0;  
        int fact = 1;  
  
        // Calcula el factorial de los números del 1 al 5.  
        for(int i = 1; i <= 5; i++) { ←  
            sum += i; // i es vista a lo largo del loop  
            fact *= i;  
        }  
  
        // Aquí, i no se ve.  
  
        Console.WriteLine("La sumatoria es " + sum);  
        Console.WriteLine("El factorial es " + fact);  
    }  
}
```

Advierte que **i** es declarada dentro del loop.

Cuando declaras una variable dentro del loop **for**, hay un aspecto importante que debes tener en mente: el alcance de la variable termina junto con la declaración **for**. (Es decir, el alcance de la variable está limitado al loop **for**), y fuera de éste la variable deja de existir. Así, en el ejemplo

anterior, **i** no es accesible fuera del loop **for**. Si necesitas utilizar la variable de control del loop en cualquier otro punto del programa, no debes declararle dentro del loop.

Antes de continuar, tal vez quieras experimentar con tus propias variaciones en el loop **for**. Como verás, es una herramienta de repetición fascinante.

## El loop **while**

Otra instrucción de repetición de C# es **while**. El formato general del loop **while** es el siguiente:

```
while(condición) declaración;
```

donde *declaración* puede ser una sola declaración o un bloque de las mismas y *condición* define la condición que controla el loop y puede ser cualquier expresión booleana válida. La declaración se realiza mientras la condición sea verdadera. Cuando la condición sea falsa, el flujo del programa pasa a la línea inmediata posterior al loop.

A continuación presentamos un ejemplo sencillo en el cual se utiliza **while** para presentar las letras del alfabeto:

```
// Muestra el uso del loop while.

using System;

class WhileDemo {
    static void Main() {
        char ch;

        // Presenta el alfabeto utilizando un loop while.
        ch = 'a';
        while(ch <= 'z') { ← Loop while ch es menor que o igual a 'z'.
            Console.Write(ch);
            ch++;
        }
    }
}
```

Aquí, la variable **ch** es inicializada con el valor de la letra *a*. En cada repetición del loop el valor de **ch** es presentado en pantalla y es incrementado. El proceso continúa hasta que **ch** es mayor que *z*.

Al igual que el loop **for**, **while** verifica la expresión condicional al inicio del loop, lo cual significa que es posible que el código del loop no se ejecute en absoluto. Por lo regular, esta característica elimina la necesidad de realizar una prueba por separado antes de ejecutar el loop. El siguiente programa ilustra tal característica del loop **while**. Calcula las potencias de 2 del 0 al 9.

```
// Calcula las potencias de 2.

using System;

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

```
int e;
int resultado;

for(int i=0; i < 10; i++) {
    resultado = 1;
    e = i;
    while (e > 0) {
        resultado *= 2; ← Esto no se ejecutará cuando
        e--;           e sea cero o negativo.
    }
}

Console.WriteLine("2 a la potencia " + i +
                  " es " + resultado);
}
```

Los datos de salida generados por el programa son los siguientes:

```
2 a la potencia 0 es 1
2 a la potencia 1 es 2
2 a la potencia 2 es 4
2 a la potencia 3 es 8
2 a la potencia 4 es 16
2 a la potencia 5 es 32
2 a la potencia 6 es 64
2 a la potencia 7 es 128
2 a la potencia 8 es 256
2 a la potencia 9 es 512
```

Advierte que el loop **while** se ejecuta sólo cuando **e** es mayor que 0. Así, cuando **e** es 0, lo que sucede en la primera repetición del loop **for**, el loop **while** se omite.

## Pregunta al experto

**P:** Dada la flexibilidad heredada en todos los loops de C#, ¿qué criterio debo utilizar para seleccionar uno? Es decir, ¿cómo puedo elegir el loop correcto para una tarea específica?

**R:** Utiliza el loop **for** cuando conozcas la cantidad exacta de repeticiones. Utiliza **do-while** cuando necesites que el loop realice por lo menos una repetición. Utiliza **while** cuando el loop realizará una cantidad desconocida de repeticiones.

## El loop do-while

El siguiente loop es **do-while**. Contrariamente a **for** y **while**, en los cuales la condición es probada al principio del loop, **do-while** la realiza al final. Esto significa que **do-while** se ejecutará por lo menos una vez. El formato general de **do-while** es el siguiente:

```
do {  
    declaraciones;  
} while(condición);
```

Aunque las llaves no son indispensables cuando se ejecuta una sola declaración, su uso ayuda a mejorar la legibilidad del constructor **do-while**, previniendo además posibles confusiones con **while**. El loop **do-while** se ejecuta mientras la expresión condicional sea verdadera.

El siguiente programa se ejecuta hasta que el usuario escribe la letra **q**:

```
// Muestra el uso del loop do-while.  
  
using System;  
  
class DWDemo {  
    static void Main() {  
        char ch;  
  
        do {  
            Console.Write("Presiona una tecla seguida de ENTER: ");  
            ch = (char) Console.Read(); // lee la tecla oprimida  
        } while (ch != 'q'); // La declaración se repite mientras  
        // ch sea diferente de 'q'.  
    }  
}
```

Utilizando el loop **do-while** podemos mejorar el programa de adivinanza que se presentó en el capítulo anterior. En esta ocasión, el programa se ejecutará una u otra vez hasta que adivines la letra.

```
// Juego adivina la letra, versión 4.  
  
using System;  
  
class Adivina4 {  
    static void Main() {  
        char ch, respuesta = 'K';  
  
        do {  
            Console.WriteLine("Estoy pensando una letra entre A y Z.");  
            Console.Write("¿La puedes adivinar?: ");  
  
            // Lee una letra, pero omite cr/lf.
```

```

do {
    ch = (char) Console.Read();
} while(ch == '\n' | ch == '\r');

if(ch == respuesta) Console.WriteLine("** ¡Correcto! **");
else {
    Console.Write("...Lo siento, estás ");
    if(ch < respuesta) Console.WriteLine("demasiado bajo");
    else Console.WriteLine("demasiado alto");
    Console.WriteLine("¡Prueba de nuevo!\n");
}
} while(respuesta != ch);
}
}

```

He aquí un ejemplo de los datos de salida:

```

Estoy pensando una letra entre A y Z.
¿La puedes adivinar?: A
...Lo siento, estás demasiado bajo
¡Prueba de nuevo!

```

```

Estoy pensando una letra entre A y Z.
¿La puedes adivinar?: Z
...Lo siento, estás demasiado alto
¡Prueba de nuevo!

```

```

Estoy pensando una letra entre A y Z.
¿La puedes adivinar?: K
** ¡Correcto! **

```

Advierte otro punto de interés en este programa. El loop **do-while** que se muestra en el ejemplo obtiene el siguiente carácter omitiendo los retornos de carro y líneas en blanco que puedan insertarse en la línea de entrada:

```

// Lee una letra, pero omite retornos de carro/líneas en blanco.
do {
    ch = (char) Console.Read(); // get a char
} while(ch == '\n' | ch == '\r');

```

Veamos por qué es necesario este loop. Como dijimos anteriormente, los datos de entrada en la consola son almacenados de manera temporal en la memoria: debes oprimir la tecla ENTER para que sean enviados al procesador. Presionar la tecla ENTER provoca que se genere un retorno de carro y una línea en blanco. Estos caracteres pasan a formar parte de la memoria temporal, pero el loop los descarta con la condición de leer los datos de entrada hasta que ninguno de los dos esté presente.

**Prueba esto****Mejorar el sistema de ayuda C#**

El siguiente programa expande el sistema de ayuda C# que comenzó a construirse en la sección anterior *Prueba esto*. Esta versión añade la sintaxis para los loops **for**, **while** y **do-while**. También verifica la selección del menú que realiza el usuario, repitiéndose hasta que escribe una opción válida.

**Paso a paso**

- 1.** Copia el código de **Ayuda.cs** a un nuevo archivo llamado **Ayuda2.cs**.
- 2.** Cambia la porción del programa que muestra las opciones y utiliza el loop que se presenta a continuación:

```
do {
    Console.WriteLine("Ayuda sobre:");
    Console.WriteLine(" 1. if");
    Console.WriteLine(" 2. switch");
    Console.WriteLine(" 3. for");
    Console.WriteLine(" 4. while");
    Console.WriteLine(" 5. do-while\n");
    Console.Write("Selecciona uno: ");
    do {
        opción = (char) Console.Read();
    } while (opción == '\n' | opción == '\r');
} while(opción < '1' | opción > '5');
```

Advierte que se utiliza un loop **do-while** anidado para descartar cualesquiera caracteres sospechosos que puedan presentarse en los datos de entrada, como retornos de carro o líneas en blanco. Después de realizar este cambio, el programa mostrará repetidamente el menú hasta que el usuario seleccione una opción válida, es decir, un número entre el 1 y el 5.

- 3.** Expande la declaración **switch** para incluir los loops **for**, **while** y **do-while**, como se muestra a continuación:

```
switch(opción) {
    case '1':
        Console.WriteLine("if:\n");
        Console.WriteLine("if(condición) declaración;");
        Console.WriteLine("else declaración;");
        break;
    case '2':
        Console.WriteLine("switch:\n");
        Console.WriteLine("switch(expresión) { ");
        Console.WriteLine("    case constante:");
        Console.WriteLine("        secuencia de declaraciones");
        Console.WriteLine("        break;");
        Console.WriteLine("    // ...");
        Console.WriteLine("}");
        break;
```

```

case '3':
    Console.WriteLine("for:\n");
    Console.Write("for(inicialización; condición; reiteración)");
    Console.WriteLine(" declaración;");
    break;
case '4':
    Console.WriteLine("while:\n");
    Console.WriteLine("while(condición) declaración;");
    break;
case '5':
    Console.WriteLine("do-while:\n");
    Console.WriteLine("do {");
    Console.WriteLine(" declaración;");
    Console.WriteLine("} while (condición);");
    break;
}

```

Advierte que no existe **default** en esta versión de **switch**. No es necesario incluir una secuencia **default** para manejar las opciones inválidas, porque el loop del menú asegura que siempre será insertada una opción válida.

#### 4. He aquí el programa completo de **Ayuda2.cs**:

```

/*
Un sistema de ayuda mejorado que utiliza un
do-while para procesar la selección del menú.
*/
using System;
class Ayuda2 {
    static void Main() {
        char opción;

        do {
            Console.WriteLine("Ayuda sobre:");
            Console.WriteLine(" 1. if");
            Console.WriteLine(" 2. switch");
            Console.WriteLine(" 3. for");
            Console.WriteLine(" 4. while");
            Console.WriteLine(" 5. do-while\n");
            Console.Write("Selecciona uno: ");
            do {
                opción = (char) Console.Read();
            } while(opción == '\n' | opción == '\r');
        } while(opción < '1' | opción > '5');

        Console.WriteLine("\n");
        switch(opción) {
            case '1':
                Console.WriteLine("if:\n");

```

(continúa)

```
Console.WriteLine("if(condición) declaración;");
Console.WriteLine("else declaración;");
break;
case '2':
    Console.WriteLine("switch:\n");
    Console.WriteLine("switch(expresión) { ");
    Console.WriteLine("    case constante:");
    Console.WriteLine("        secuencia de declaraciones");
    Console.WriteLine("        break;");
    Console.WriteLine("    // ...");
    Console.WriteLine("} ");
    break;
case '3':
    Console.WriteLine("for:\n");
    Console.WriteLine("for(inicialización; condición; reiteración)");
    Console.WriteLine("    declaración");
    break;
case '4':
    Console.WriteLine("while:\n");
    Console.WriteLine("while(condición) declaración");
    break;
case '5':
    Console.WriteLine("do-while:\n");
    Console.WriteLine("do { ");
    Console.WriteLine("    declaración");
    Console.WriteLine("} while (condición);");
    break;
}
}
```

## Usar break para salir de un loop

Es posible forzar la terminación inmediata de un loop, omitiendo cualquier código restante en su cuerpo y la prueba condicional del mismo, utilizando la declaración **break**. Cuando una declaración **break** es descubierta dentro del loop, éste se da por terminado y el flujo del programa continúa en la declaración inmediata posterior. He aquí un ejemplo:

```
// Usar break para salir de un loop.

using System;

class BreakDemo {
    static void Main() {
        int num;
```

```

num = 100;

// Se repite mientras el cuadrado de i sea menor a num.
for(int i=0; i < num; i++) {

    // Termina el loop si i*i >= 100.
    if(i*i >= num) break; ← Usa break para terminar el loop.

    Console.WriteLine(i + " ");
}
Console.WriteLine("Loop completado.");
}
}

```

El programa genera los siguientes datos de salida:

```
0 1 2 3 4 5 6 7 8 9 Loop completado.
```

Como puedes ver, aunque el loop **for** está diseñado para ejecutarse de 0 a **num** (que en este caso es 100), la declaración **break** hace que el loop se interrumpa con anticipación, cuando el cuadrado de **i** es mayor que o igual a **num**.

La declaración **break** puede utilizarse con cualquier loop de C#, incluyendo las repeticiones al infinito declaradas de manera intencional. Por ejemplo, el siguiente programa simplemente lee los datos de entrada hasta que el usuario presiona la tecla **q**:

```

// Lee los datos de entrada hasta que recibe una q.

using System;

class Break2 {
    static void Main() {
        char ch;

        for( ; ; ) { ←
            ch = (char) Console.Read(); ← Este loop infinito es interrumpido por break.
            if(ch == 'q') break; ←
        }
        Console.WriteLine("Presionaste q!");
    }
}

```

Cuando se usa dentro de un conjunto de loops anidados, la declaración **break** interrumpirá sólo la declaración más profunda. Por ejemplo:

```

// Uso de break en loops anidados.

using System;

class Break3 {
    static void Main() {

```

```
for(int i=0; i<3; i++) {
    Console.WriteLine("Cuenta de loop externo: " + i);
    Console.Write("    Cuenta del loop interno: ");

    int t = 0;
    while(t < 100) {
        if(t == 10) break; ← Break interrumpe el loop más profundo.
        Console.Write(t + " ");
        t++;
    }
    Console.WriteLine();
}
Console.WriteLine("Loop completado.");
}
```

El programa genera los siguientes datos de salida:

```
Cuenta de loop externo: 0
    Cuenta del loop interno: 0 1 2 3 4 5 6 7 8 9
Cuenta de loop externo: 1
    Cuenta del loop interno: 0 1 2 3 4 5 6 7 8 9
Cuenta de loop externo: 2
    Cuenta del loop interno: 0 1 2 3 4 5 6 7 8 9
Loop completado.
```

Como puedes ver, la declaración **break** en el loop más profundo provoca sólo la interrupción del mismo. El loop externo no se ve afectado.

He aquí otros dos puntos importantes a recordar sobre **break**. Primero, en un loop puede aparecer más de una declaración **break**, pero debes tener cuidado. Demasiadas declaraciones **break** tienden a romper la estructura de tu código. Segundo, el **break** que interrumpe una declaración **switch** afecta sólo esa declaración, pero no a los loops encerrados en ella.

### Pregunta al experto

**P:** Sé que en Java las declaraciones **break** y **continue** pueden ser utilizadas con una etiqueta. ¿C# soporta esa misma característica?

**R:** No. Los diseñadores de C# no les dieron a **break** y **continue** esa capacidad. En vez de eso, ambas declaraciones funcionan de la misma manera como lo hacen en C y C++. Una de las razones por las que C# no siguió la línea trazada por Java en este aspecto es porque Java no soporta la declaración **goto**, pero C# sí lo hace. Por tales razones, Java necesitaba dar a **break** y **continue** poder extra, para compensar la carencia de **goto**.

## Utilizar continue

Es posible forzar el aumento en la velocidad de repetición de un loop omitiendo su estructura normal. Esto se realiza utilizando la declaración **continue**, que obliga a que se realice la siguiente repetición del loop, omitiendo cualquier código que se encuentre en medio. Así, **continue** es el complemento esencial de **break**. Por ejemplo, en el siguiente programa se utiliza **continue** para ayudar a presentar sólo los números pares entre 0 y 100:

```
// Uso de continue.

using System;

class ParesDemo {
    static void Main() {
        int i;

        // Presenta los números pares entre 0 y 100.
        for(i = 0; i<=100; i++) {
            // Repite si i es par.
            if((i%2) != 0) continue; ← Repite el loop si i es par.

            Console.WriteLine(i);
        }
    }
}
```

Sólo se presentan los números pares porque los nones cumplen con la condición para que se ejecute **continue** y por ello omiten la invocación de la declaración **WriteLine()**.

En los loops **while** y **do-while**, una declaración **continue** hará que el controlador vaya directamente a la expresión condicional. En el caso de **for**, la expresión de reiteración del loop se evalúa y luego se ejecuta la expresión condicional.

Los buenos usos de **continue** son escasos. Una de las razones es que C# proporciona un rico conjunto de declaraciones de repetición que se pueden aplicar a la mayor parte de los programas. Sin embargo, para aquellas circunstancias especiales en las que se necesita forzar la repetición, la declaración **continue** proporciona un método estructurado para realizar la tarea.

## La instrucción goto

**Goto** es la declaración de trayectoria sin condiciones de C#. Cuando localiza una, el flujo del programa se traslada a la localización especificada por **goto**. Esta declaración cayó de la gracia de los programadores hace muchos años, porque alentaba la creación de “código espagueti”, una revolución de saltos sin condición específica que daban por resultado un código muy difícil de leer. Sin embargo, **goto** aún se utiliza, y en ocasiones con efectividad. En este libro no haremos un juicio sobre su validez como controlador del flujo de un programa, pero debemos dejar muy en claro que en la programación no existe una situación que requiera su uso, no es un elemento que se necesite para tener un lenguaje de programación completo. En vez de ello, **goto** es una declaración conveniente; si se utiliza con inteligencia puede resultar de provecho en algunas circunstancias. Por ello, **goto** no se utiliza fuera de esta sección. La principal preocupación que muestran los programadores respecto a **goto** es su tendencia a desordenar un programa y hacerlo casi ilegible. Sin embargo,

existen algunas circunstancias en las cuales el uso de **goto** sirve para clarificar el programa en vez de confundirlo.

La declaración **goto** requiere una etiqueta para su operación. Una *etiqueta* es un identificador C# válido seguido por un punto y coma. Más aún, la etiqueta debe colocarse en el mismo método donde se encuentra el **goto** que la utiliza. Por ejemplo, una repetición del 1 al 100 puede escribirse utilizando un **goto** y una etiqueta, como se muestra aquí:

```
x = 1;
repetición1: ←
    x++;
    if(x < 100) goto repetición1; → El flujo del programa se traslada a repetición1.
```

Un buen uso de **goto** es salir de una rutina profundamente anidada. He aquí un ejemplo sencillo:

```
// Muestra un uso de goto.

using System;

class Uso_goto {
    static void Main() {
        int i=0, j=0, k=0;

        for(i=0; i < 10; i++) {
            for(j=0; j < 10; j++) {
                for(k=0; k < 10; k++) {
                    Console.WriteLine("i, j, k: " + i + " " + j + " " + k);
                    if(k == 3) goto detener;
                }
            }
        }
    }
    detener:
        Console.WriteLine("¡Detenido! i, j, k: " + i + ", " + j + ", " + k);
    }
}
```

Los datos de salida generados por el programa:

```
i, j, k: 0 0 0
i, j, k: 0 0 1
i, j, k: 0 0 2
i, j, k: 0 0 3
¡Detenido! i, j, k: 0, 0, 3
```

Eliminar el uso de **goto** requeriría forzosamente la aplicación de tres declaraciones **if** e igual cantidad de **break**. En este caso particular el uso de **goto** simplifica el código. Recuerda que éste es un ejemplo inventado que sirve como ilustración; ahora puedes imaginar situaciones aplicables al mundo real donde **goto** represente un auténtico beneficio.

**Goto** tiene una restricción importante: no puede utilizarse para trasladar el flujo del programa dentro de un bloque. Por supuesto, puede salir del bloque, como lo muestra el ejemplo anterior.

Además de trabajar con etiquetas “normales”, **goto** puede utilizarse para trasladarse a una etiqueta **case** o a una **default** dentro de una declaración **switch**. Por ejemplo, la siguiente es una declaración **switch** válida:

```
switch (x) {
    case 1: // ...
        goto default;
    case 2: // ...
        goto case 1;
    default: // ...
        break;
}
```

La instrucción **goto default** hace que el flujo del programa se traslade a la etiqueta **default**. La instrucción **goto case 1** realiza la misma acción hacia **case 1**. Dada la restricción antes mencionada, no es posible trasladar el flujo a la mitad de **switch** desde afuera, porque **switch** define un bloque; por ello, este tipo de declaraciones **goto** deben ejecutarse desde el interior de **switch**.

## Prueba esto Finaliza el programa de ayuda C#

Es tiempo de dar los toques finales al sistema de ayuda de C#. Esta versión añade la sintaxis para **break**, **continue** y **goto**. También permite que el usuario solicite la sintaxis correcta para más de una declaración. Esto se realiza añadiendo un loop externo que se ejecuta hasta que el usuario oprime la tecla **q**.

### Paso a paso

1. Copia **Ayuda2.cs** a un nuevo archivo llamado **Ayuda3.cs**.
2. Envuelve todo el código del programa con un loop infinito **for**. Interrumpe este loop, utilizando **break**, cuando el usuario presione la tecla **q**. Como este loop envuelve todo el código del programa, interrumpirlo equivale a dar por terminado todo el programa.
3. Cambia el loop del menú como se muestra a continuación:

```
do {
    Console.WriteLine("Ayuda sobre:");
    Console.WriteLine(" 1. if");
    Console.WriteLine(" 2. switch");
    Console.WriteLine(" 3. for");
    Console.WriteLine(" 4. while");
    Console.WriteLine(" 5. do-while");
    Console.WriteLine(" 6. break");
    Console.WriteLine(" 7. continue");
    Console.WriteLine(" 8. goto\n");
    Console.Write("Selecciona una (q para terminar): ");
    do {
        opción = (char) Console.Read();
    } while(opción == '\n' | opción == '\r');
} while(opción < '1' | opción > '8' & opción != 'q');
```

(continúa)

Observa que ahora este loop incluye las declaraciones **break**, **continue** y **goto**. También acepta **q** como una opción válida.

4. Expande la declaración **switch** para incluir las declaraciones **break**, **continue** y **goto**, como se muestra aquí:

```
case '6':
    Console.WriteLine("break:\n");
    Console.WriteLine("break;");
    break;
case '7':
    Console.WriteLine("continue:\n");
    Console.WriteLine("continue;");
    break;
case '8':
    Console.WriteLine("goto:\n");
    Console.WriteLine("goto label;");
    break;
```

5. He aquí el programa **Ayuda3.cs** completo:

```
/*
    El sistema de ayuda para declaraciones C#
    completo, que procesa múltiples solicitudes.
*/

using System;

class Ayuda3 {
    static void Main() {
        char opción;

        for( ; ; ) {
            do {
                Console.WriteLine("Ayuda sobre:");
                Console.WriteLine(" 1. if");
                Console.WriteLine(" 2. switch");
                Console.WriteLine(" 3. for");
                Console.WriteLine(" 4. while");
                Console.WriteLine(" 5. do-while");
                Console.WriteLine(" 6. break");
                Console.WriteLine(" 7. continue");
                Console.WriteLine(" 8. goto\n");
                Console.Write("Selecciona una (q para terminar): ");
                do {
                    opción = (char) Console.Read();
                } while(opción == '\n' | opción == '\r');
            } while( opción < '1' | opción > '8' & opción != 'q');

            if(opción == 'q') break;
        }
    }
}
```

```
Console.WriteLine("\n");

switch(opción) {
    case '1':
        Console.WriteLine("The if:\n");
        Console.WriteLine("if(condición) declaración;");
        Console.WriteLine("else declaración;");
        break;
    case '2':
        Console.WriteLine("The switch:\n");
        Console.WriteLine("switch(expresión) { ");
        Console.WriteLine("    case constante:");
        Console.WriteLine("        secuencia de declaraciones");
        Console.WriteLine("        break;");
        Console.WriteLine("    // ...");
        Console.WriteLine("} ");
        break;
    case '3':
        Console.WriteLine("The for:\n");
        Console.WriteLine("for(inicialización; condición; reiteración)");
        Console.WriteLine("    declaración");
        break;
    case '4':
        Console.WriteLine("The while:\n");
        Console.WriteLine("while(condición) declaración");
        break;
    case '5':
        Console.WriteLine("The do-while:\n");
        Console.WriteLine("do { ");
        Console.WriteLine("    declaración");
        Console.WriteLine("} while (condición);");
        break;
    case '6':
        Console.WriteLine("The break:\n");
        Console.WriteLine("break");
        break;
    case '7':
        Console.WriteLine("The continue:\n");
        Console.WriteLine("continue");
        break;
    case '8':
        Console.WriteLine("The goto:\n");
        Console.WriteLine("goto label");
        break;
}
Console.WriteLine();
}
```

(continúa)

He aquí un ejemplo de su ejecución:

Ayuda sobre:

- 1. if
- 2. switch
- 3. for
- 4. while
- 5. do-while
- 6. break
- 7. continue
- 8. goto

Selecciona una (q para terminar): 1

if:

```
if(condición) declaración;  
else declaración;
```

Ayuda sobre:

- 1. if
- 2. switch
- 3. for
- 4. while
- 5. do-while
- 6. break
- 7. continue
- 8. goto

Selecciona una (q para terminar): 6

break:

```
break;
```

Ayuda sobre:

- 1. if
- 2. switch
- 3. for
- 4. while
- 5. do-while
- 6. break
- 7. continue
- 8. goto

Selecciona una (q para terminar): q

---

## Loops anidados

Como has visto en ejemplos anteriores, un loop puede anidarse dentro de otro. La anidación de loops es utilizada para resolver una gran variedad de problemas en los programas y son una parte esencial de la programación. Por ello, antes de abandonar el tema de las declaraciones de repetición en C#, veamos un ejemplo más de anidación de loops. El siguiente programa utiliza un loop **for** anidado para encontrar los factores de los números del 2 al 100:

```
// Uso de loops anidados para encontrar factores de los números entre el  
2 y el 100.  
  
using System;  
  
class Factores {  
    static void Main() {  
  
        for(int i=2; i <= 100; i++) {  
            Console.Write("Factores de " + i + ": ");  
            for(int j = 2; j <= i/2; j++)  
                if((i%j) == 0) Console.Write(j + " ");  
            Console.WriteLine();  
        }  
    }  
}
```

He aquí una porción de los datos de salida generados por el programa:

```
Factores de 2:  
Factores de 3:  
Factores de 4: 2  
Factores de 5:  
Factores de 6: 2 3  
Factores de 7:  
Factores de 8: 2 4  
Factores de 9: 3  
Factores de 10: 2 5  
Factores de 11:  
Factores de 12: 2 3 4 6  
Factores de 13:  
Factores de 14: 2 7  
Factores de 15: 3 5  
Factores de 16: 2 4 8  
Factores de 17:  
Factores de 18: 2 3 6 9  
Factores de 19:  
Factores de 20: 2 4 5 10
```

En el programa, el loop externo ejecuta **i** del 2 al 100. El loop interno verifica sucesivamente todos los números de 2 a **i**, y presenta en pantalla aquellos que son divididos exactamente entre **i**.

 **Autoexamen Capítulo 3**

1. Escribe un programa que lea caracteres del teclado hasta que reciba un punto. Haz que el programa cuente el número de espacios vacíos. Reporta el total al término del programa.
2. En la declaración **switch**, ¿puede el código pasar en secuencia de un **case** a otro?
3. Escribe el formato general de una escalera **if-else-if**.
4. Dado el siguiente código:

```
if(x < 10)
    if(y > 100) {
        if(!done) x = z;
        else y = z;
    }
else Console.WriteLine("error"); // qué tal si...
```

¿con cuál **if** está asociado el último **else**?

5. Escribe la declaración **for** para un loop que cuente de 1 000 a 0 a intervalos de -2.
6. ¿Es válido el siguiente fragmento?

```
for(int i = 0; i < num; i++)
    sum += i;

count = i;
```

7. Explica la función de **break**.
8. En el siguiente fragmento, después de que la declaración **break** se ejecuta, ¿qué se muestra en pantalla?

```
for(i = 0; i < 10; i++) {
    while(running) {
        if(x<y) break;
        // ...
    }
    Console.WriteLine("después del while");
}
Console.WriteLine("después del for");
```

9. ¿Qué presenta en pantalla el siguiente fragmento?

```
for(int i = 0; i<10; i++) {
    Console.Write(i + " ");
    if((i%2) == 0) continue;
    Console.Writeline();
}
```

- 10.** La expresión de reiteración en un loop **for** no siempre altera la variable de control del loop con una magnitud fija. En lugar de ello, la variable de control del loop puede cambiar de manera arbitraria. Utilizando este concepto, escribe un programa que utilice un loop **for** para generar y mostrar la progresión 1, 2, 4, 8, 16, 32 y así sucesivamente.
- 11.** Las letras minúsculas del código ASCII están separadas de las mayúsculas por 32. Así, para convertir una minúscula en mayúscula debes restarle 32. Utiliza esta información para escribir un programa que lea caracteres del teclado. Haz que convierta todas las minúsculas en mayúsculas, todas las mayúsculas en minúsculas y que las presente en pantalla. No hagas ningún cambio más a ningún otro carácter. Haz que el programa se detenga cuando el usuario presione la tecla del punto. Al final, haz que el programa muestre la cantidad de cambios que realizó.



# Capítulo 4

Introducción a las  
clases, objetos y  
métodos

## Habilidades y conceptos clave

- Fundamentos de las clases
  - Instanciar un objeto
  - Conceptos básicos del método
  - Conceptos básicos de los parámetros
  - Obtener un valor de un método
  - Constructores
  - **new**
  - Recolector de basura
  - Destructores
  - La palabra clave **this**
- 

**A**ntes de avanzar más en el estudio de C#, necesitas obtener conocimiento sobre las clases. La clase es la esencia de C# porque define la naturaleza de un objeto; y como tal, la clase conforma las bases de la programación orientada a objetos en C#. Dentro de una clase se definen los datos y el código que actúa sobre ellos. El código está contenido en métodos. Como las clases, los objetos y los métodos son fundamentales para C#, en este capítulo presentamos una introducción sobre ellos. Tener un conocimiento básico de estas características te permitirá escribir programas más sofisticados y entender mejor ciertos elementos clave de C# descritos en el capítulo 5.

## Fundamentos de las clases

Hemos utilizado clases desde el principio de este libro. Por supuesto, sólo se han utilizado clases extremadamente simples, y no hemos aprovechado la mayoría de sus características. Como verás, las clases son mucho más poderosas que los ejemplos limitados que se han presentado hasta el momento.

Comencemos por revisar los fundamentos. Una clase es una plantilla con la que se define la forma de un objeto. Típicamente especifica tanto el código como los datos, donde el primero actúa sobre los segundos. C# utiliza una especificación de clase para construir *objetos*. Los objetos son *instancias* de una clase. Así, una clase es, en esencia, un conjunto de planos que especifican cómo construir un objeto. Es muy importante que quede clara esta cuestión: una clase es una abstracción lógica; no es sino hasta el momento en que se crea un objeto de esa clase cuando existe una representación física de la misma en la memoria de la computadora.

Otro punto importante: recuerda que los métodos y las variables que constituyen la clase son llamados *miembros* de clase.

## La forma general de una clase

Cuando defines una clase, declaras los datos que contiene y el código que opera sobre ellos. Mientras que una clase simple puede contener sólo código o sólo datos, la mayoría de las clases en el mundo real contienen ambos.

En términos generales, los datos están contenidos en variables de instancia definidas por la clase, y el código está contenido en los métodos. Es importante dejar en claro, sin embargo, que las definiciones de C# incluyen una amplia gama de miembros de clase, entre los que se cuentan variables de instancia, variables estáticas, constantes, métodos, constructores, destructores, indexadores, eventos, operadores y propiedades. Por el momento limitaremos nuestro análisis de las clases a sus elementos esenciales: variables de instancia y métodos. Más tarde, en este mismo capítulo, analizaremos los constructores y los destructores. Los demás miembros de clase son analizados en capítulos subsecuentes.

Una clase se crea utilizando la palabra clave **class**. El formato general para definir una **class** que contiene sólo variables de instancia y métodos es el siguiente:

```
class nombre-de-clase {  
    // Declaración de variables de instancia.  
    acceso tipo var1;  
    acceso tipo var2;  
    // ...  
    acceso tipo varN;  
  
    // Declaración de métodos.  
    acceso tipo-recuperado método1(parámetros) {  
        // cuerpo del método  
    }  
    acceso tipo-recuperado método2(parámetros) {  
        // cuerpo del método  
    }  
    // ...  
    acceso tipo-recuperado métodoN(parámetros) {  
        // cuerpo del método  
    }  
}
```

Observa que cada variable y método es antecedido por un *acceso*. Aquí, *acceso* es una especificación, como **public**, que indica la manera como puede ser accedido ese miembro de la clase. Como se mencionó en el capítulo 1, los miembros de clase pueden ser exclusivos de una clase, o bien tener más alcance. El delimitador de acceso determina el alcance permitido para la variable o método al que se aplica. El delimitador de acceso es opcional, y en su ausencia, el exclusivo de la clase a la que pertenece. Los miembros con acceso exclusivo (**private**) pueden ser utilizados únicamente por otros miembros de la misma clase. Para los ejemplos de este capítulo, todos los miembros de clase (excepto el método **Main()**) serán especificados como **public** (públicos). Ello significa que podrán ser utilizados por todo el código restante, incluso el escrito fuera de la clase a la que pertenecen. El método **Main()** continuará utilizando el acceso por omisión porque es la

práctica actualmente recomendada. Regresaremos al tema de los delimitadores de acceso en un capítulo posterior, cuando ya conozcas los aspectos fundamentales de las clases.

Una clase bien diseñada debe definir una y sólo una entidad lógica, aunque no existe una regla sintáctica que obligue a hacerlo. Por ejemplo, una clase que almacena nombres y números telefónicos, por lo regular no se utilizará para almacenar información sobre el mercado de acciones, el promedio de precipitación pluvial, los ciclos de las manchas solares o cualquier otra información sin relación lógica con los primeros. El punto aquí es que una clase bien formada agrupa información lógicamente relacionada. Colocar información incoherente en la misma clase ¡destruirá rápidamente la estructura de tu código!

Hasta ahora, las clases que hemos estado utilizando tienen un solo método: **Main( )**. Pronto aprenderás a utilizar otros. Sin embargo, observa que el formato general de una clase no especifica la necesidad de un método **Main( )**, porque éste se requiere sólo si la clase donde aparece es el punto de inicio de tu programa.

## Definir una clase

Para ilustrar las clases desarrollaremos una que encapsule información sobre vehículos, como automóviles, camionetas y camiones. Esta clase se llama **Vehículo**, y almacenará tres elementos informativos sobre un vehículo: el número de pasajeros que puede transportar, su capacidad de combustible y el promedio de consumo de este último (en kilómetros por litro).

La primera versión de **Vehículo** se muestra a continuación. Define tres variables de instancia: **Pasajeros**, **CapComb** y **Kpl**. Observa que **Vehículo** no tiene métodos; en este momento es una clase sólo con datos; en posteriores secciones se aumentarán los métodos.

```
class Vehículo {  
    public int Pasajeros; // cantidad de pasajeros  
    public int CapComb; // la capacidad de combustible en litros  
    public int Kpl; // consumo de combustible en kilómetros por litro  
}
```

Las variables de instancia definidas en **Vehículo** ilustran la manera como se declaran en general. El formato estándar para declarar una variable de instancia es el siguiente:

*acceso tipo nombre-de-variable;*

Aquí, *acceso* especifica el alcance de la variable, *tipo* es el tipo de dato asignado a la variable y *nombre-de-variable* es el nombre con el que será conocida en el programa. De esta manera, fuera del delimitador de acceso, una variable de instancia se declara de la misma manera que una local. Para **Vehículo**, las variables van antecedidas por el delimitador **public**. Como se explicó anteriormente, esto permite que sean accesadas por el código que se encuentra fuera de la clase **Vehículo**.

Una definición de clase (**class**) crea un nuevo tipo de dato. En este caso, el nuevo tipo de dato recibe el nombre de **Vehículo**. Utilizarás este nombre para declarar objetos de tipo **Vehículo**. Recuerda que la declaración de una **clase** es sólo un tipo descriptivo, no crea un objeto de hecho. Así, el código anterior no trae a la existencia ningún objeto del tipo **Vehículo**.

Para crear de hecho un objeto **Vehículo**, utilizarás una declaración como la siguiente:

```
Vehículo minivan = new Vehículo(); // crea un objeto de Vehículo llamado  
minivan
```

Después de que se ejecuta esta declaración, **minivan** será una instancia de **Vehículo**. De esta manera tendrá una existencia “física”. Por el momento, no te preocupes por los detalles de esta declaración.

Cada vez que creas una instancia de una clase, estás creando un objeto que contiene su propia copia de cada variable de instancia definida en la clase. Así, cada objeto de **Vehículo** contendrá sus propias copias de las variables **Pasajeros**, **CapComb** y **Kpl**. Para accesar estas variables, utilizarás el operador de acceso a miembros de la clase, que es un punto (.); comúnmente se le conoce como *operador de punto*. Este operador vincula el nombre de un objeto con el nombre de un miembro. El formato general del operador de punto es el siguiente:

*objeto.miembro*

De tal manera que el objeto es especificado a la izquierda del punto y el miembro a la derecha. Por ejemplo, para asignar el valor 72 a la variable **CapComb** de **minivan**, utiliza la siguiente declaración:

```
minivan.CapComb = 72;
```

En general, puedes utilizar el operador de punto para obtener acceso tanto a variables como a métodos.

He aquí el programa completo que utiliza la clase **Vehículo**:

```
/* Un programa que utiliza la clase Vehículo.  
Llama a este archivo UsaVehículo.cs  
*/  
  
using System;  
  
// Una clase que encapsula información sobre vehículos.  
class Vehículo {  
    public int Pasajeros; // cantidad de pasajeros  
    public int CapComb; // la capacidad de combustible en litros  
    public int Kpl; // consumo de combustible en kilómetros por litro  
}  
  
// Esta clase declara un objeto de tipo Vehículo.  
class VehículoDemo {  
    static void Main() {  
        Vehículo minivan = new Vehículo(); // Crea una instancia de Vehículo llamada minivan.  
        int alcance;  
  
        // Asigna valores a los campos en minivan.  
        minivan.Pasajeros = 7; // Observa el uso del operador de punto para obtener acceso a un miembro de la clase.  
        minivan.CapComb = 72;  
        minivan.Kpl = 5;  
  
        // Calcula el alcance de kilómetros por tanque lleno.  
        alcance = minivan.CapComb * minivan.Kpl;  
  
        Console.WriteLine("Minivan puede transportar " + minivan.Pasajeros +  
                          " un alcance de " + alcance);  
    }  
}
```

Este programa consiste de dos clases: **Vehículo** y **VehículoDemo**. Dentro de la segunda, el método **Main()** crea una instancia de la primera llamada **minivan**. El código dentro de **Main()** accesa las variables de instancia asociadas con **minivan**, asignándoles sus respectivos valores y utilizándolos. Es importante comprender que **Vehículo** y **VehículoDemo** son dos clases separadas. La única relación que tienen entre sí es que una de ellas crea una instancia de la otra. Aunque se trata de clases diferentes, el código dentro de **VehículoDemo** puede accesar los miembros de **Vehículo** porque han sido declarados **públicos** (con la palabra clave **public**). Si no tuvieran el delimitador **public**, su acceso estaría restringido a la clase **Vehículo**, y **VehículoDemo** no podría hacer uso de ellos.

Suponiendo que le diste al archivo el nombre sugerido, **UsaVehículo.cs**, al compilar el programa se creará un archivo llamado **UsaVehículo.exe**. Ambas clases, **Vehículo** y **VehículoDemo**, pasan a formar parte del archivo ejecutable automáticamente. El programa muestra los siguientes datos de salida:

```
Minivan puede transportar 7 un alcance de 360
```

No es necesario que ambas clases, **Vehículo** y **VehículoDemo**, se localicen en el mismo archivo de código fuente. Es posible colocar cada clase en su propio archivo, que llevarían los nombres de **Vehículo.cs** y **VehículoDemo.cs**, por ejemplo. Sólo tienes que indicar al compilador de C# que ambos archivos están vinculados. Por ejemplo, si dividiste el programa en dos piezas de acuerdo con lo indicado anteriormente, podrías utilizar esta línea de comando para compilar el programa:

```
csc Vehículo.cs VehículoDemo.cs
```

Con este procedimiento se creará el archivo **VehículoDemo.exe**, porque ésa es la clase que contiene el método **Main()**. Si utilizas el IDE de Visual Studio, necesitarás añadir ambos archivos al mismo proyecto y luego generarlos.

Antes de continuar, revisemos un principio fundamental: cada objeto tiene sus propias copias de las variables de instancia definidas por su clase. De esta forma, el contenido de las variables en un objeto puede ser diferente del contenido de las mismas variables en otro objeto. No hay conexión entre los dos objetos, salvo que ambos pertenezcan al mismo tipo. Por ejemplo, si tienes dos objetos **Vehículo**, cada uno tiene su propia copia de las variables **Pasajeros**, **CapComb** y **Kpl**, pero su contenido puede ser diferente en cada uno de ellos. El siguiente programa muestra este hecho:

```
// Este programa crea dos objetos Vehículo.  
  
using System;  
  
// Una clase que encapsula información sobre vehículos.  
class Vehículo {  
    public int Pasajeros; // cantidad de pasajeros  
    public int CapComb; // la capacidad de combustible en litros  
    public int Kpl; // consumo de combustible en kilómetros por litro  
}  
  
// Esta clase declara dos objetos de tipo Vehículo.  
class DosVehículos {
```

```

static void Main() {
    Vehículo minivan = new Vehículo();
    Vehículo deportivo = new Vehículo(); ← Recuerda que minivan y deportivo hacen
                                              referencia a dos objetos distintos.

    int alcance1, alcance2;

    // Asignar valores a los campos en minivan.
    minivan.Pasajeros = 7;
    minivan.CapComb = 72;
    minivan.Kpl = 5;

    // Asignar valores a los campos en deportivo.
    deportivo.Pasajeros = 2;
    deportivo.CapComb = 63;
    deportivo.Kpl = 3;

    // Calcula el alcance de kilómetros por tanque lleno.
    alcance1 = minivan.CapComb * minivan.Kpl;
    alcance2 = deportivo.CapComb * deportivo.Kpl;

    Console.WriteLine("Minivan puede transportar " + minivan.Pasajeros +
                      " un alcance de " + alcance1);

    Console.WriteLine("Deportivo puede transportar " + deportivo.
                      Pasajeros + " un alcance de " + alcance2);
}
}

```

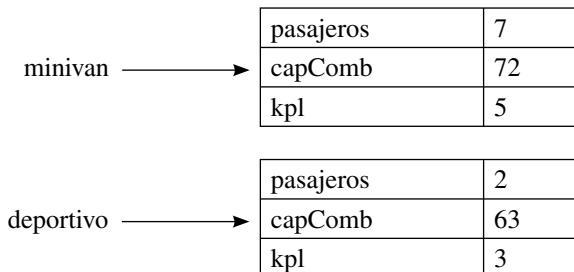
Los datos de salida generados por este programa son los siguientes:

```

Minivan puede transportar 7 un alcance de 360
Deportivo puede transportar 2 un alcance de 189

```

Como puedes ver, los datos de **minivan** están completamente separados de los datos contenidos en **deportivo**. La figura 4-1 ilustra esta situación:



**Figura 4-1** Las variables de instancia de un objeto están separadas de las del otro

## Cómo se crean los objetos

En los ejemplos anteriores se utilizó la siguiente línea para crear un objeto tipo **Vehículo**:

```
Vehículo minivan = new Vehículo();
```

Esta declaración realiza tres funciones: primero, declara una variable llamada **minivan** de la clase tipo **Vehículo**. Esta variable no es, en sí, un objeto. En lugar de ello, es simplemente una variable que puede *hacer referencia a* un objeto. Segundo, la declaración crea de hecho una instancia física del objeto. Esto se lleva a cabo utilizando el operador **new**. Finalmente, asigna a **minivan** una referencia a ese objeto. Así, después de que se ejecuta la línea, **minivan** hace referencia a un objeto de tipo **Vehículo**.

El operador **new** *coloca dinámicamente* (esto es, en tiempo de ejecución) el objeto en la memoria de la computadora y regresa una referencia al mismo. Esta referencia es almacenada entonces en una variable. Así, todos los objetos de clase en C# deben ser colocados en memoria dinámicamente.

Como podrías esperar, es posible separar la declaración **minivan** de la creación del objeto al cual hará referencia, como se muestra aquí:

```
Vehículo minivan; // declara la referencia a un objeto  
minivan = new Vehículo(); // coloca en memoria el objeto Vehículo
```

La primera línea declara **minivan** como referencia a un objeto tipo **Vehículo**. De esta manera, **minivan** es una variable que puede hacer referencia a un objeto, pero no es un objeto en sí. La siguiente línea crea un nuevo objeto **Vehículo** y asigna a la variable **minivan** una referencia al mismo. Ahora **minivan** está vinculada con un objeto.

El hecho de que los objetos de clase sean accesados a través de una referencia explica por qué las clases son llamadas *tipos de referencia*. La diferencia clave entre los tipos de valor y los tipos de referencia es el significado intrínseco de cada uno. En una variable tipo valor, la variable en sí contiene el valor que le ha sido asignado. Por ejemplo, en el código:

```
int x;  
x = 10;
```

**x** contiene el valor 10, porque es una variable **int**, tipo de dato para almacenar valores. Sin embargo, en el caso de

```
Vehículo minivan = new Vehículo();
```

**minivan** no contiene en sí el objeto, sólo contiene una referencia al mismo.

## Variables de referencia y asignaciones

En una operación de asignación, las variables de referencia actúan de manera diferente en comparación con las variables de valor, como **int**. Cuando asignas una variable de valor a otra, la situación es muy clara: la variable del lado izquierdo recibe una *copia del valor* de la variable del lado derecho. Cuando asignas una variable de referencia a otra, la situación es un poco más compleja, porque provocas que la variable del lado izquierdo haga referencia al objeto que, a su vez, es re-

ferido por la variable del lado derecho. El efecto de esta diferencia puede traer algunos resultados poco intuitivos. Por ejemplo, considera el siguiente fragmento de código:

```
Vehículo carro1 = new Vehículo();
Vehículo carro2 = carro1;
```

A primera vista, es fácil suponer que **carro1** y **carro2** hacen referencia a dos objetos distintos, pero no es así. Todo lo contrario, **carro1** y **carro2** hacen referencia al *mismo* objeto. La asignación de **carro1** a **carro2** simplemente hace que **carro2** haga referencia al mismo objeto que **carro1**. De esta manera, el objeto puede ser modificado a través de **carro1** o **carro2** de manera indistinta. Por ejemplo, después de que se ejecuta esta asignación:

```
carro1.Kpl = 26;
```

los datos de salida generados por los siguientes métodos **WriteLine( )** mostrarán el mismo valor, 26:

```
Console.Writeline(carro1.Kpl);
Console.Writeline(carro2.Kpl);
```

**Carro1** y **carro2** hacen referencia al mismo objeto y no están vinculados de ninguna otra manera. Por ejemplo, una subsecuente asignación a **carro2** simplemente cambia el objeto al que hace referencia. Por ejemplo:

```
Vehículo carro1 = new Vehículo();
Vehículo carro2 = carro1;
Vehículo carro3 = new Vehículo();

carro2 = carro3; //ahora, carro2 y carro3 hacen referencia al mismo
objeto.
```

Después de que se ejecuta esta secuencia, **carro2** hace referencia al mismo objeto que **carro3**. Por supuesto, **carro1** sigue haciendo referencia a su objeto original.

## Métodos

Como se explicó, las variables de instancia y los métodos son dos de los principales elementos integrantes de las clases. Hasta este momento, la clase **Vehículo** contiene datos, pero no métodos. Aunque las clases que contienen sólo datos son perfectamente válidas, la mayoría de las clases también contendrán métodos. Los métodos son subrutinas que manipulan los datos definidos por las clases y, en muchos casos, proporcionan acceso a los mismos. Por lo regular, otras partes de tu programa interactuarán con una clase a través de sus métodos.

Un método contiene una o más declaraciones. En el código C# bien escrito, cada método realiza sólo una tarea. Cada método tiene un nombre, que es utilizado para invocarlo. En general puedes nombrar un método utilizando cualquier identificador válido que sea de tu agrado. Sin embargo, debes recordar que el nombre **Main()** está reservado para el método que inicia la ejecución de tu programa. Tampoco debes utilizar palabras clave de C# como nombre de método.

Para diferenciar los métodos en el cuerpo del texto, en este libro hemos usado y seguiremos usando la convención que se ha hecho común para escribir sobre C#: un método tendrá una pareja de paréntesis después de su nombre. Por ejemplo, si el nombre de un método es **ObtenerValor**,

será escrito **ObtenerValor()** cuando su nombre sea utilizado en una oración. Esta notación te ayudará a distinguir los nombres de las variables de los nombres de los métodos en el texto.

A continuación presentamos el formato general de un método:

```
acceso tipo-respuesta nombre(lista-de-parámetros) {  
    // cuerpo del método  
}
```

Aquí, *acceso* es un delimitador que establece cuáles partes del programa pueden invocar este método en particular. Como se explicó anteriormente, es opcional. De no declararse, el método pasa a ser exclusivo de la clase en la que es declarado. Hasta ahora hemos declarado métodos públicos (**public**) con el fin de que puedan ser invocados por cualquier otra parte del código en el programa. La parte *tipo-respuesta* especifica el tipo de dato que regresará el método. Puede ser cualquier tipo válido, incluyendo los tipos clases que tú mismo programas. Si el método no regresa ningún valor, su tipo de respuesta debe ser **void** (vacío). El nombre del método se especifica en el lugar correspondiente a *nombre*; puede ser cualquier identificador legal que no provoque conflicto dentro del espacio de la declaración actual. La parte correspondiente a *lista-de-parámetros* es una secuencia de parejas tipo-identificador separadas por comas. Los parámetros son esencialmente variables que reciben el valor de los *argumentos* que se transmiten al método cuando éste es invocado. Si el método no tiene parámetros, la lista debe quedar vacía.

## Añadir un método a la clase Vehículo

Como se acaba de explicar, por lo regular los métodos de una clase manipulan y proporcionan acceso a los datos de la misma. Con esto en mente, recuerda que en los anteriores ejemplos **Main()** calcula una aproximación de los kilómetros que recorre un vehículo multiplicando el consumo de combustible promedio por su capacidad de almacenamiento. Aunque técnicamente correcto, no es la mejor manera de manejar esta tarea. Lo ideal es manejar el cálculo mencionado en la clase **Vehículo**. La razón es fácil de comprender: los kilómetros que recorre un vehículo dependen de la capacidad del tanque de combustible y de su promedio de consumo, y ambas cantidades están encapsuladas en la clase **Vehículo**. Al añadirle un método que haga el cálculo, estarás enriqueciendo su estructura orientada a objetos.

Para añadir un método a **Vehículo**, debes especificarlo dentro de su declaración. Por ejemplo, la siguiente versión de **Vehículo** contiene un método llamado **Alcance()** que muestra el aproximado de los kilómetros que recorre:

```
// Añadir alcance a Vehículo.  
  
using System;  
  
// Una clase que encapsula información sobre vehículos.  
class Vehículo {  
    public int Pasajeros; // cantidad de pasajeros  
    public int CapComb; // la capacidad de combustible en litros  
    public int Kpl; // consumo de combustible en kilómetros por litro  
  
    // Muestra el alcance.  
    public void Alcance() { ← El método Alcance() está contenido  
        // ...  
    }  
}
```

```

        Console.WriteLine("El alcance es " + CapComb * Kpl);
    }
}

class NuevoMétodo {
    static void Main() {
        Vehículo minivan = new Vehículo();
        Vehículo deportivo = new Vehículo();

        // Asigna valores a los campos en minivan.
        minivan.Pasajeros = 7;
        minivan.CapComb = 72;
        minivan.Kpl = 5;

        // Asigna valores a los campos en deportivo.
        deportivo.Pasajeros = 2;
        deportivo.CapComb = 63;
        deportivo.Kpl = 3;

        Console.Write("Minivan puede transportar " + minivan.Pasajeros +
                     ". ");
        minivan.Alcance(); // muestra alcance de minivan

        Console.Write("Deportivo puede transportar " + deportivo.Pasajeros +
                     ". ");
        deportivo.Alcance(); // muestra alcance de deportivo.
    }
}

```

Observa que **CapComb** y **Kpl** se utilizan directamente, sin el operador de punto.

Este programa genera los siguientes datos de salida:

```

Minivan puede transportar 7. El alcance es 360
Deportivo puede transportar 2. El alcance es 189

```

Veamos los elementos clave de este programa, comenzando con el método **Alcance()**. La primera línea de **Alcance()** es:

```
public void Alcance() {
```

Esta línea declara un método llamado **Alcance()** que no tiene parámetros. Está especificado como **public**, por lo que puede ser utilizado por todas las demás partes del programa. Su tipo de respuesta es **void**, por lo que **Alcance()** no regresará ningún valor a la parte del programa que lo invoca. La línea finaliza con una llave de apertura, que da inicio al cuerpo del método.

El cuerpo de **Alcance()** está formado por una sola línea:

```
Console.WriteLine("El alcance es " + CapComb * Kpl);
```

Esta declaración muestra el alcance de desplazamiento del vehículo multiplicando **CapComb** por **Kpl**. Pero cada objeto aplica sus propios valores a la multiplicación, almacenados en sus respectivas variables **CapComb** y **Kpl**, y hace un cálculo diferente. Por ello aparecen dos resultados, uno para **minivan** y otro para **deportivo**.

El método **Alcance()** termina cuando aparece su llave de clausura. Esto provoca que el control del programa regrese al elemento que realizó la invocación.

A continuación, observa con mucha atención esta línea de código localizada dentro de **Main()**:

```
minivan.Alcance();
```

Esta declaración invoca el método **Alcance()** en **minivan**. Es decir, invoca el **Alcance()** relativo al objeto referido por **minivan**, utilizando el operador de punto. Cuando se invoca un método, el control del programa es transferido al método invocado. Cuando el método termina, el control es transferido de regreso al invocador, y la ejecución del programa continúa en la línea inmediata posterior a la invocación.

En este caso, la invocación a **minivan.Alcance()** envía a pantalla el alcance del vehículo definido por **minivan**. De manera similar, la invocación a **deportivo.Alcance()** muestra el alcance del vehículo definido por **deportivo**. Cada vez que se invoca **Alcance()**, muestra el alcance definido por el objeto que lo invoca.

Hay algo muy importante que debes observar dentro del método **Alcance()**: las variables de instancia **CapComb** y **Kpl** son referidas directamente, sin el uso del operador de punto. Cuando un método utiliza una variable de instancia definida en su misma clase, lo hace directamente, sin hacer referencia explícita a un objeto y sin utilizar el operador de punto. La razón es fácil de comprender si lo piensas un poco: un método siempre es invocado respecto a un objeto de su clase. Una vez que ha ocurrido la invocación, el objeto es conocido. Así, dentro de un método, no hay necesidad de especificar el objeto por segunda ocasión. Esto significa que **CapComb** y **Kpl** dentro de **Alcance()** hacen referencia implícita a las copias de esas variables que se encuentran en el objeto que hace la invocación de **Alcance()**.

## Regresar de un método

En general, existen dos condiciones que provocan la finalización de un método y el regreso del flujo del programa a la parte invocadora. La primera condición, como lo muestra el método **Alcance()** del ejemplo anterior, sucede cuando aparece la llave de clausura del método. La segunda es cuando se ejecuta una declaración **return**. Existen dos formatos de **return**: uno para ser utilizado por los métodos **void** (aquellos que no regresan ningún valor como respuesta) y otro para los que sí envían un valor de respuesta. El primer formato se muestra a continuación, el segundo es explicado en la siguiente sección.

En un método **void**, puedes provocar la terminación inmediata de un método utilizando el siguiente formato de **return**:

```
return;
```

Cuando se ejecuta esta declaración, el control del programa regresa a la parte invocadora, omitiendo todo el código restante dentro del método. Por ejemplo, analiza el siguiente método:

```
public void MiMétodo() {
    int i;

    for(i=0; i<10; i++) {
        if(i == 5) return; // detiene el loop en 5
        Console.WriteLine(i);
    }
}
```

Aquí, el loop **for** sólo se ejecutará de 0 a 5, porque una vez que **i** alcance el valor de 5, el método regresará el control a la parte invocadora.

Está permitido tener múltiples declaraciones de **return** en un método, en especial cuando hay dos o más rutas para abandonarlo. Por ejemplo:

```
public void MiMétodo() {
    // ...
    if(terminado) return;
    // ...
    if(error) return;
}
```

Aquí, el método regresa el control cuando haya terminado o bien cuando ocurra un error. Sin embargo, debes tener cuidado, porque tener muchas salidas de un método puede romper la estructura de tu código, por lo que debes evitar utilizarlas a la ligera.

En resumen: un método **void** puede regresar el control del programa de dos maneras: cuando aparece su llave de clausura o cuando se ejecuta una declaración **return**.

## Regresar un valor como respuesta

Aunque los métodos cuyo tipo de regreso es **void** no son raros, la mayoría regresarán un valor como respuesta. De hecho, la capacidad para regresar un valor es una de las características más útiles de un método. Ya has visto un ejemplo de un método que regresa un valor: cuando utilizamos **Math.Sqrt()** para obtener una raíz cuadrada en los capítulos 2 y 3.

En programación, regresar valores se utiliza para una amplia variedad de propósitos. En algunos casos, como **Math.Sqrt()**, el valor regresado contiene el resultado de un cálculo. En otros casos simplemente indica éxito o fracaso. En otros más puede contener código de estatus. Cualquiera que sea el propósito, utilizar los valores regresados por un método es una parte integral de la programación en C#.

Los métodos regresan un valor a la rutina invocadora utilizando este formato de **return**:

```
return valor;
```

Aquí, *valor* es, por supuesto, el valor regresado.

Puedes utilizar un valor regresado para mejorar la implementación de **Alcance()**. En lugar de mostrar en pantalla el alcance del vehículo, un táctica mejor es hacer que **Alcance()** haga el cálculo y regrese el valor obtenido. Entre las ventajas que presenta este procedimiento es que puedes utilizar el valor para realizar otros cálculos. El siguiente ejemplo modifica **Alcance()** para regresar el valor en lugar de mostrarlo en pantalla:

```
// Usar un valor regresado por un método.

using System;

// Una clase que encapsula información sobre vehículos.
class Vehículo {
    public int Pasajeros; // cantidad de pasajeros
    public int CapComb; // la capacidad de combustible en litros
    public int Kpl; // consumo de combustible en kilómetros por litro
```

```
// Regresar el alcance.
public int Alcance() {
    return Kpl * CapComb; ← Ahora Alcance() regresa un valor.
}
}

class RetMet {
    static void Main() {
        Vehículo minivan = new Vehículo();
        Vehículo deportivo = new Vehículo();

        int alcance1, alcance2;

        // Asigna valores a los campos en minivan.
        minivan.Pasajeros = 7;
        minivan.CapComb = 72;
        minivan.Kpl = 5;

        // Asigna valores a los campos en deportivo.
        deportivo.Pasajeros = 2;
        deportivo.CapComb = 63;
        deportivo.Kpl = 3;

        // Obtiene los alcances.
        alcance1 = minivan.Alcance(); ← Asigna el valor regresado por
        alcance2 = deportivo.Alcance(); Alcance() a una variable.

        Console.WriteLine("Minivan transporta " + minivan.Pasajeros +
                           " con un alcance de " + alcance1 + " kilómetros.");
        Console.WriteLine("Deportivo transporta " + deportivo.Pasajeros +
                           " con un alcance de " + alcance2 + " kilómetros.");
    }
}
```

Los datos de salida que presenta el programa son:

Minivan puede transportar 7 con un alcance de 360 kilómetros.  
Deportivo puede transportar 2 con un alcance de 189 kilómetros.

En el programa, advierte que cuando se invoca **Alcance()**, se coloca del lado derecho de una declaración de asignación. Del lado izquierdo se encuentra la variable que recibirá el valor regresado por **Alcance()**. Así, después de que se ejecuta la línea,

```
alcance1 = minivan.Alcance();
```

el alcance del objeto **minivan** se almacena en **alcance1**.

Observa que ahora **Alcance()** tiene un tipo **int** de regreso. Esto significa que regresará un valor entero al invocador. El tipo de regreso de un método es importante porque el tipo de dato regresado

por un método debe ser compatible con el tipo de regreso especificado por el método. De esta manera, si quieras que un método regrese un tipo de dato **double**, su tipo de regreso debe ser **double**.

Aunque el programa anterior es correcto, no está escrito con la eficiencia que podría tener. Específicamente, no hay necesidad de tener las variables **alcance1** y **alcance2**. Es posible utilizar una invocación a **Alcance()** utilizando la declaración **WriteLine()** directamente, como se muestra a continuación:

```
Console.WriteLine("Minivan puede transportar " + minivan.Pasajeros +
    " con alcance de " + minivan.Alcance() + " kilómetros");
```

En este caso, cuando se ejecuta **WriteLine()**, **minivan.Alcance()** es invocado automáticamente y su valor será transmitido a **WriteLine()**. Más aún, puedes utilizar una invocación a **Alcance()** cada vez que se requiera el alcance de un objeto **Vehículo**. Por ejemplo, la siguiente declaración compara los alcances de dos vehículos:

```
if(v1.Alcance() > v2.Alcance()) Console.WriteLine("v1 tiene más
alcance");
```

## Pregunta al experto

**P:** He escuchado que C# detecta “código inalcanzable”. ¿Qué significa eso?

**R:** Has escuchado bien. El compilador de C# presentará un mensaje de error si creas un método que contiene código que ninguna ruta de acceso de ejecución podrá alcanzar. Observa el siguiente ejemplo:

```
public void m() {
    char a, b;

    // ...

    if(a==b) {
        Console.WriteLine("iguales");
        return;
    } else {
        Console.WriteLine("desiguales");
        return;
    }
    Console.WriteLine("esto es inalcanzable");
}
```

Aquí, el método **m()** siempre regresará el control antes de que se ejecute la última declaración **WriteLine()**. Si intentas compilar este método, recibirás un mensaje de advertencia. Por lo regular el código inalcanzable es un error del programador, por lo que es buena idea tomar en serio este tipo de advertencias.

## Usar parámetros

Es posible transmitir uno o más valores a un método cuando éste es invocado. Como se mencionó anteriormente, un valor transmitido a un método recibe el nombre de *argumento*. Dentro del método, la variable que recibe el argumento es llamada *parámetro formal*, o simplemente *parámetro*, para abbreviar. Los parámetros son declarados dentro de los paréntesis posteriores al nombre del método. La sintaxis para declarar parámetros es la misma que se utiliza para las variables. El alcance de un parámetro es el cuerpo del método que lo contiene. Fuera de su tarea especial para recibir argumentos, actúan como cualquier otra variable local.

A continuación presentamos un ejemplo sencillo que utiliza parámetros. Dentro de la clase **ChkNum**, el método **EsPar()** regresa un valor **true** (verdadero) si el valor que se le transmite es un número par. En cualquier otro caso regresa un valor **false** (falso).

Por tanto, **EsPar()** regresa datos tipo **bool**.

```
// Un ejemplo sencillo que utiliza parámetros.

using System;

// Esta clase contiene el método EsPar, que recibe un parámetro.
class ChkNum {
    // Regresa un valor true si x es par.
    public bool EsPar(int x) { ← Aquí, x es un parámetro entero de EsPar().
        if((x%2) == 0) return true;
        else return false;
    }
}

class ParamDemo {
    static void Main() {
        ChkNum e = new ChkNum(); ↓ Transmite el argumento a EsPar().
        if(e.EsPar(10)) Console.WriteLine("10 es par.");
        if(e.EsPar(9)) Console.WriteLine("9 es par.");
        if(e.EsPar(8)) Console.WriteLine("8 es par.");
    }
}
```

Los datos de salida generados por el programa:

```
10 es par.
8 es par.
```

En el programa, **EsPar()** es invocado tres veces, y en cada ocasión le es transmitido un valor diferente. Veamos de cerca este proceso. Primero, observa cómo se invoca **EsPar()**. El argumento se especifica entre los paréntesis. Cuando **EsPar()** se invoca por primera vez, transmite el valor 10. Así, cuando **EsPar()** comienza a ejecutarse, el parámetro **x** recibe el valor de 10. En la segunda invocación, el argumento es 9, por lo que el valor de **x** es igual a 9. En la tercera invocación, el

argumento es 8, que es el mismo valor que recibe `x`. Lo importante aquí es que el valor transmitido como argumento cuando se invoca `EsPar()` es el mismo que recibe su parámetro `x`.

Un método puede tener más de un parámetro. Simplemente separa uno de otro con comas. Por ejemplo, la clase **Factor** define un método llamado `EsFactor()` que determina si el primer parámetro es factor del segundo.

```
using System;

class Factor {
    // Determina si x es un factor de y.
    public bool EsFactor(int x, int y) { ←———— Este método tiene dos parámetros.
        if((y % x) == 0) return true;
        else return false;
    }
}

class EsFac {
    static void Main() {
        Factor x = new Factor();

        if(x.EsFactor(2, 20)) Console.WriteLine("2 es factor");
        if(x.EsFactor(3, 20)) Console.WriteLine("esto no se mostrará");
    }
}
```

Observa que cuando se invoca `EsFactor()`, los argumentos también están separados por comas.

Cuando se utilizan múltiples parámetros, cada uno de ellos especifica su propio tipo de dato, que puede ser diferente en cada uno de ellos. Por ejemplo, la siguiente línea es perfectamente válida:

```
int MiMétodo(int a, double b, float c) {
// ...
```

## Añadir un método con parámetros a vehículo

Puedes utilizar un método parametrizado para añadir nuevas características a la clase **Vehículo**: la capacidad de calcular la cantidad de combustible necesario para recorrer cierta distancia. Este nuevo método recibe el nombre de `CombNecesario()`. Este método toma la cantidad de kilómetros que quieras recorrer y regresa la cantidad de litros requeridos. El método `CombNecesario()` se define de la siguiente manera:

```
public double CombNecesario(int kilómetros) {
    return (double) kilómetros / Kpl;
}
```

Observa que este método regresa un valor de tipo **double**. Esto es útil porque es posible que la cantidad de combustible necesario para recorrer cierta distancia no sea un número entero.

## 136 Fundamentos de C# 3.0

A continuación presentamos la clase **Vehículo** completa que incluye el método **CombNecesario()**:

```
/*
    Añade un método parametrizado que calcula
    el combustible requerido para recorrer cierta distancia.
*/

using System;

class Vehículo {
    public int Pasajeros; // cantidad de pasajeros
    public int CapComb; // la capacidad de combustible en litros
    public int Kpl; // consumo de combustible en kilómetros por litro

    // Regresa el alcance.
    public int Alcance() {
        return Kpl * CapComb;
    }

    // Calcular el combustible necesario para recorrer cierta distancia.
    public double CombNecesario(int kilómetros) { ← Usa el método parametrizado
        return (double) kilómetros / Kpl;
    }
}

class CapComb {
    static void Main() {
        Vehículo minivan = new Vehículo();
        Vehículo deportivo = new Vehículo();
        double litros;
        int dist = 252;

        // Asignar valores a los campos en minivan.
        minivan.Pasajeros = 7;
        minivan.CapComb = 72;
        minivan.Kpl = 5;

        // Asignar valores a los campos en deportivo.
        deportivo.Pasajeros = 2;
        deportivo.CapComb = 63;
        deportivo.Kpl = 3;

        litros = minivan.CombNecesario(dist);

        Console.WriteLine("Para viajar " + dist + " kilómetros minivan
            necesita " + litros + " litros de combustible.");

        litros = deportivo.CombNecesario(dist);
    }
}
```

```

        Console.WriteLine("Para viajar " + dist + " kilómetros deportivo
                          necesita " + litros + " litros de combustible.");
    }
}

```

Los datos de salida que genera el programa son:

```

Para viajar 252 kilómetros minivan necesita 50.4 litros de combustible.
Para viajar 252 kilómetros deportivo necesita 84 litros de combustible.

```

## Prueba esto

## Crear una clase Ayuda

Si uno intentara resumir la esencia de la clase en una sola frase, tal vez sería: una clase encapsula funcionalidad. Por supuesto, algunas veces el truco consiste en saber cuándo termina una “funcionalidad” y comienza otra. Como regla general, tus clases deben ser los bloques constructores de aplicaciones más grandes. Para hacer esto, cada clase debe representar una sola unidad funcional que realiza acciones claramente delineadas. En este sentido, tus clases deben ser lo más pequeñas posible, ¡pero no demasiado! Es decir, las clases que contienen funcionalidad extrema son confusas y desestabilizan la estructura del código, pero las clases que contienen muy poca funcionalidad están fragmentadas. ¿Cuál es el punto de equilibrio? Es aquí donde la *ciencia* de la programación se convierte en *arte*. Por fortuna, la mayoría de los programadores encuentran que el balance resulta más fácil de reconocer con la experiencia.

Para comenzar a adquirir esa experiencia, convertiremos el sistema de ayuda desarrollado en la sección *Prueba esto* del capítulo 3 en una clase Ayuda. Veamos por qué es una buena idea. Primero, el sistema de ayuda define una sola unidad lógica. Simplemente muestra la sintaxis de las declaraciones de control de C#. Por ello su funcionalidad es compacta y bien definida. Segundo, poner el sistema de ayuda en una clase es un placer estético. Cuando quieras ofrecer el sistema de ayuda a un usuario, simplemente tendrás que hacer un objeto. Finalmente, dado que la ayuda está encapsulada, puede actualizarse o modificarse sin provocar efectos secundarios indeseados en el programa que la utilice.

## Paso a paso

1. Crea un nuevo archivo llamado **AyudaClaseDemo.cs**. Para ahorrarte algo de captura tal vez quieras copiar el archivo de la última sección *Prueba esto* del capítulo 3, **Ayuda3.cs**, al nuevo archivo.
2. Para convertir el sistema de ayuda en una clase, primero debes determinar con precisión las partes que integran al sistema de ayuda. Por ejemplo, en **Ayuda3.cs**, hay código para: mostrar un menú, ingresar la opción del usuario, verificar una respuesta válida y mostrar la información sobre el tema seleccionado. El programa también hace un loop hasta que el usuario oprime la tecla *q*. Si lo piensas, es claro que el menú, la verificación de una respuesta válida y mostrar la información son partes integrales del sistema de ayuda; al contrario de cómo se obtienen los datos de entrada del usuario y la manera de procesar la repetición de solicitudes. De esta manera, crearás una clase que muestre la información de ayuda, el menú y que verifique una selección válida. Esta funcionalidad puede organizarse en métodos, llamados **AyudaDe()**, **MuestraMenú()** y **EsVálido()**.

(continúa)

**3.** Crea el método **AyudaDe()**, como se muestra aquí:

```
public void AyudaDe(char que) {
    switch(que) {
        case '1':
            Console.WriteLine("if:\n");
            Console.WriteLine("if(condición) declaración;");
            Console.WriteLine("else declaración;");
            break;
        case '2':
            Console.WriteLine("switch:\n");
            Console.WriteLine("switch(expresión) { ");
            Console.WriteLine("    case constante:");
            Console.WriteLine("        secuencia de declaraciones");
            Console.WriteLine("        break;");
            Console.WriteLine("    // ...");
            Console.WriteLine("} ");
            break;
        case '3':
            Console.WriteLine("for:\n");
            Console.WriteLine("for(inicialización; condición; reiteración)");
            Console.WriteLine("    declaración");
            break;
        case '4':
            Console.WriteLine("while:\n");
            Console.WriteLine("while(condición) declaración");
            break;
        case '5':
            Console.WriteLine("do-while:\n");
            Console.WriteLine("do { ");
            Console.WriteLine("    declaración");
            Console.WriteLine("} while (condición);");
            break;
        case '6':
            Console.WriteLine("break:\n");
            Console.WriteLine("break");
            break;
        case '7':
            Console.WriteLine("continue:\n");
            Console.WriteLine("continue");
            break;
        case '8':
            Console.WriteLine("goto:\n");
            Console.WriteLine("goto etiqueta;");
            break;
    }
    Console.WriteLine();
}
```

**4.** Crea el método **MuestraMenú()**:

```
public void MuestraMenú() {
    Console.WriteLine("Ayuda de:");
    Console.WriteLine(" 1. if");
    Console.WriteLine(" 2. switch");
    Console.WriteLine(" 3. for");
    Console.WriteLine(" 4. while");
    Console.WriteLine(" 5. do-while");
    Console.WriteLine(" 6. break");
    Console.WriteLine(" 7. continue");
    Console.WriteLine(" 8. goto\n");
    Console.Write("Seleccione una (q para salir): ");
}
```

**5.** Crea el método **EsVálido()**, como se muestra aquí:

```
public bool EsVálido(char ch) {
    if(ch < '1' | ch > '8' & ch != 'q') return false;
    else return true;
}
```

**6.** Ensambla los métodos anteriores a la clase **Ayuda**, como se muestra a continuación:

```
class Ayuda {
    public void AyudaDe(char que) {
        switch(que) {
            case '1':
                Console.WriteLine("if:\n");
                Console.WriteLine("if(condición) declaración;");
                Console.WriteLine("else declaración;");
                break;
            case '2':
                Console.WriteLine("switch:\n");
                Console.WriteLine("switch(expresión) {}");
                Console.WriteLine("  case constante:");
                Console.WriteLine("    secuencia de declaraciones");
                Console.WriteLine("    break;");
                Console.WriteLine("  // ...");
                Console.WriteLine("}");
                break;
            case '3':
                Console.WriteLine("for:\n");
                Console.WriteLine("for(inicialización; condición; reiteración)");
                Console.WriteLine("  declaración");
                break;
            case '4':
                Console.WriteLine("while:\n");
                Console.WriteLine("while(condición) declaración");
                break;
        }
    }
}
```

(continúa)

```
        case '5':
            Console.WriteLine("do-while:\n");
            Console.WriteLine("do {");
            Console.WriteLine(" declaración;");
            Console.WriteLine("} while (condición);");
            break;
        case '6':
            Console.WriteLine("break:\n");
            Console.WriteLine("break;");
            break;
        case '7':
            Console.WriteLine("continue:\n");
            Console.WriteLine("continue; ");
            break;
        case '8':
            Console.WriteLine("goto:\n");
            Console.WriteLine("goto etiqueta; ");
            break;
    }
    Console.WriteLine();
}

public void MuestraMenú() {
    Console.WriteLine("Ayuda de:");
    Console.WriteLine(" 1. if");
    Console.WriteLine(" 2. switch");
    Console.WriteLine(" 3. for");
    Console.WriteLine(" 4. while");
    Console.WriteLine(" 5. do-while");
    Console.WriteLine(" 6. break");
    Console.WriteLine(" 7. continue");
    Console.WriteLine(" 8. goto\n");
    Console.Write("Seleccione una (q para salir): ");
}

public bool EsVálido(char ch) {
    if(ch < '1' | ch > '8' & ch != 'q') return false;
    else return true;
}

}
```

7. Finalmente, crea una clase llamada **AyudaClaseDemo.cs** que utilice la nueva clase **Ayuda**. Haz que **Main()** muestre la información de ayuda hasta que el usuario oprima la tecla *q*. A continuación aparece el código completo de **AyudaClaseDemo.cs**:

```
// El sistema de Ayuda del capítulo 3 convertido en la clase Ayuda.

using System;
```

```
class Ayuda {
    public void AyudaDe(char que) {
        switch(que) {
            case '1':
                Console.WriteLine("if:\n");
                Console.WriteLine("if(condición) declaración;");
                Console.WriteLine("else declaración;");
                break;
            case '2':
                Console.WriteLine("switch:\n");
                Console.WriteLine("switch(expresión) {");
                Console.WriteLine("    case constante:");
                Console.WriteLine("        secuencia de declaraciones");
                Console.WriteLine("        break;");
                Console.WriteLine("    // ...");
                Console.WriteLine("} ");
                break;
            case '3':
                Console.WriteLine("for:\n");
                Console.WriteLine("for(inicialización; condición; reiteración)");
                Console.WriteLine("    declaración");
                break;
            case '4':
                Console.WriteLine("while:\n");
                Console.WriteLine("while(condición) declaración");
                break;
            case '5':
                Console.WriteLine("do-while:\n");
                Console.WriteLine("do {");
                Console.WriteLine("    declaración");
                Console.WriteLine("} while (condición);");
                break;
            case '6':
                Console.WriteLine("break:\n");
                Console.WriteLine("break");
                break;
            case '7':
                Console.WriteLine("continue:\n");
                Console.WriteLine("continue");
                break;
            case '8':
                Console.WriteLine("goto:\n");
                Console.WriteLine("goto etiqueta");
                break;
        }
        Console.WriteLine();
    }
}
```

(continúa)

```
// Muestra menú de ayuda.
public void MuestraMenú() {
    Console.WriteLine("Ayuda de:");
    Console.WriteLine(" 1. if");
    Console.WriteLine(" 2. switch");
    Console.WriteLine(" 3. for");
    Console.WriteLine(" 4. while");
    Console.WriteLine(" 5. do-while");
    Console.WriteLine(" 6. break");
    Console.WriteLine(" 7. continue");
    Console.WriteLine(" 8. goto\n");
    Console.Write("Seleccione una (q para salir): ");
}

// Prueba para una selección de menú válida.
public bool EsVálido(char ch) {
    if(ch < '1' | ch > '8' & ch != 'q') return false;
    else return true;
}

class AyudaClaseDemo {
    static void Main() {
        char choice;
        Ayuda objayuda = new Ayuda();

        for(;;) {
            do {
                objayuda.MuestraMenú();
                do {
                    choice = (char) Console.Read();
                } while (choice == '\n' | choice == '\r');

                } while( !objayuda.EsVálido(choice) );

                if(choice == 'q') break;

                Console.WriteLine("\n");

                objayuda.AyudaDe(choice);
            }
        }
}
```

Cuando ejecutes el programa, notarás que su funcionalidad es la misma que la versión final del capítulo 3. La ventaja de esta técnica es que ahora tienes un componente de sistema de ayuda, que podrás reutilizar cuando sea necesario.

## Constructores

En los ejemplos anteriores, las variables de instancia de cada objeto **Vehículo** tuvieron que ser capturadas manualmente, utilizando una secuencia de declaraciones como:

```
minivan.Pasajeros = 7;
minivan.CapComb = 72;
minivan.Kpl = 5;
```

Un procedimiento como éste nunca sería utilizado en un código C# escrito profesionalmente. Además de ser susceptible a errores (puedes olvidar escribir uno de los campos), simplemente existe una mejor manera de realizar la tarea: el constructor.

Un *constructor* inicializa un objeto cuando es creado. Tiene el mismo nombre que su clase y su sintaxis es parecida a la del método. Sin embargo, un constructor no tiene un tipo de dato específico para su regreso. El formato general de un constructor es el siguiente:

```
acceso nombre-de-clase(lista-de-parámetros) {
    // código del constructor
}
```

Por lo regular, utilizarás un constructor para establecer el valor inicial de las variables de instancia definidas por la clase, o para iniciar cualquier otro procedimiento de arranque requerido para crear un objeto completamente formado. Muy a menudo el *acceso* es **public** porque un constructor suele ser invocado desde el exterior de la clase a la que pertenece. La *lista de parámetros* puede estar vacía o puede especificar uno o más parámetros.

Todas las clases tienen constructores, ya sea que lo definas o dejes de hacerlo, porque C# proporciona en automático un constructor por omisión que inicializa todos los miembros de la clase con sus respectivos valores por defecto. Para la mayoría de los tipos de valor el inicio por defecto es cero; para los tipos **bool** es **false**; para los tipos de referencia el valor por defecto es nulo. Sin embargo, una vez que defines tu propio constructor, el de C# por omisión deja de utilizarse.

A continuación presentamos un ejemplo que utiliza un constructor:

```
// Un constructor sencillo.

using System;

class MiClase {
    public int x;

    public MiClase() {
        x = 10;           ← El constructor para MiClase.
    }
}

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

```
        MiClase t1 = new MiClase();
        MiClase t2 = new MiClase();

        Console.WriteLine(t1.x + " " + t2.x);
    }
}
```

En este ejemplo, el constructor de **MiClase** es

```
public MiClase() {
    x = 10;
}
```

Observa que el constructor está especificado como **public** porque será invocado por código definido fuera de su clase. Como se mencionó, la mayoría de los constructores son declarados **public** por esa razón. El constructor del ejemplo asigna el valor 10 a la variable de instancia **x** de **MiClase**. El constructor es invocado por la palabra clave **new** cuando se crea un objeto. Por ejemplo, en la línea

```
MiClase t1 = new MiClase();
```

el constructor **MiClase()** es invocado en el objeto **t1**, dándole a **t1.x** el valor de 10. Lo mismo sucede con **t2**. Después de su construcción, **t2.x** también tiene un valor igual 10. Así, los datos de salida son

```
10 10
```

## Constructores parametrizados

En el ejemplo anterior se utilizó un constructor sin parámetros. Si bien esto es correcto para algunas situaciones, la mayoría de las veces necesitarás un constructor que acepte uno o más parámetros. Los parámetros son añadidos al constructor de la misma manera como son agregados a un método: simplemente decláralos dentro de los paréntesis que se localizan después del nombre del constructor. En el siguiente ejemplo, **MiClase** cuenta con un constructor parametrizado:

```
// Un constructor parametrizado.

using System;

class MiClase {
    public int x;

    public MiClase(int i) {
        x = i;
    }
}

class ParmConsDemo {
    static void Main() {
        MiClase t1 = new MiClase(10);
```

← Este constructor tiene un parámetro.

```

        MiClase t2 = new MiClase(88);

        Console.WriteLine(t1.x + " " + t2.x);
    }
}

```

Los datos de salida de este programa son:

10 88

En esta versión del programa, el constructor **MiClase()** define un parámetro llamado **i**, que es utilizado para inicializar la variable de instancia **x**. Así, cuando se ejecuta la línea

```
MiClase t1 = new MiClase(10);
```

se transmite el valor 10 a **i**, que luego es asignada a **x**.

## Añadir un constructor a la clase Vehículo

Podemos mejorar la clase **Vehículo** añadiendo un constructor que inicialice automáticamente los campos **Pasajeros**, **CapComb** y **Kpl** cuando se construya un objeto. Pon especial atención a la manera como son creados los objetos **Vehículo**.

```

// Añade un constructor a Vehículo.

using System;

class Vehículo {
    public int Pasajeros; // cantidad de pasajeros
    public int CapComb; // capacidad de combustible en litros
    public int Kpl; // consumo de combustible en kilómetros por litro

    // Éste es un constructor para Vehículo.
    public Vehículo(int p, int f, int m) {
        Pasajeros = p;
        CapComb = f;
        Kpl = m;
    }

    // Regresa el alcance.
    public int Alcance() {
        return Kpl * CapComb;
    }

    // Calcula el combustible necesario para recorrer cierta distancia.
    public double CombNecesario(int kilómetros) {
        return (double) kilómetros / Kpl;
    }
}

```

← Constructor para **Vehículo**.

```
class VehConsDemo {  
    static void Main() {  
  
        // Construye vehículos completos.  
        Vehículo minivan = new Vehículo(7, 72, 5);  
        Vehículo deportivo = new Vehículo(2, 63, 3); ← Pasa información a  
        double litros;  
        int dist = 252;  
  
        litros = minivan.CombNecesario(dist);  
  
        Console.WriteLine("Para recorrer " + dist + " kilómetros una minivan  
                          necesita " + litros + " litros de combustible.");  
  
        litros = deportivo.CombNecesario(dist);  
  
        Console.WriteLine("Para recorrer " + dist + " kilómetros un deportivo  
                          necesita " + litros + " litros de combustible.");  
  
    }  
}
```

Tanto **minivan** como **deportivo** fueron inicializados por el constructor **Vehículo()** cuando fueron creados. Cada objeto se inicializa de acuerdo con lo especificado en los parámetros de su constructor. Por ejemplo, en la siguiente línea:

```
Vehículo minivan = new Vehículo(7, 72, 5);
```

los valores 7, 72 y 5 son transmitidos al constructor **Vehículo()** cuando **new** crea el objeto. Así, **minivan** tiene una copia de **Pasajeros**, **CapComb** y **Kpl** que contienen los valores 7, 72 y 5, respectivamente. Por lo mismo, los datos de salida generados por este programa son los mismos que en la versión previa.

## El operador new revisado

Ahora que ya conoces más sobre las clases y sus constructores, veamos más de cerca el operador **new**. Su formato general es el siguiente:

```
new nombre-clase(lista-de-argumentos)
```

Aquí, *nombre-clase* es el nombre de la clase que dará origen al nuevo objeto. El nombre de la clase seguido por paréntesis especifica el constructor de la clase, como se describió en la sección anterior. Si una clase no define su propio constructor, **new** utilizará el constructor por omisión proporcionado por C#.

Como la memoria de la computadora es finita, cabe la posibilidad de que **new** no pueda asignar memoria para el nuevo objeto si ésta es insuficiente. En esos casos ocurrirá una excepción en tiempo de ejecución. (Aprenderás más sobre excepciones en el capítulo 9.) Para los programas de ejemplo que aparecen en este libro, no es necesario que te preocupes por la falta de memoria, pero necesitas considerar esta posibilidad para los programas del mundo real que escribas.

## Pregunta al experto

**P:** ¿Por qué no es necesario utilizar new para las variables de tipo valor, como int o float?

**R:** En C#, una variable de tipo valor contiene su propio valor. La memoria que lo almacena es proporcionada de manera automática cuando el programa se ejecuta. Así, no es necesario asignar de manera explícita esta porción de memoria utilizando **new**. Por el contrario, una variable de referencia almacena la referencia a un objeto y la memoria que almacena tal objeto es asignada dinámicamente durante su ejecución.

Separar los tipos fundamentales, como **int** o **char**, de los tipos de referencia mejora de manera considerable el rendimiento de tu programa. Cuando se utilizan tipos de referencia, hay una capa de rodeo que se satura con el acceso a cada objeto. Esta saturación es evadida por los tipos de valor.

Como dato interesante, diremos que está permitido utilizar **new** con los tipos de valor, como se muestra aquí:

```
int i = new int();
```

Al hacerlo, se invoca el constructor por omisión del tipo **int**, que inicializa **i** en cero. En general, aplicar **new** para un tipo de valor invoca el constructor por omisión de ese tipo. Sin embargo, no asigna dinámicamente una proporción de la memoria. Sinceramente, la mayoría de los programadores no utilizan **new** con los tipos de valor.

## Recolección de basura y destructores

Como has visto, los objetos son asignados dinámicamente a partir de un cúmulo de memoria disponible utilizando el operador **new**. Por supuesto, la memoria no es infinita y la cantidad disponible puede agotarse. En este sentido, es posible que **new** falle porque no dispone de memoria suficiente para crear el objeto deseado. Por esta razón, uno de los componentes clave del esquema de asignación dinámica es la liberación de memoria de objetos sin usar, dejándola disponible para una reasignación subsiguiente. En muchos lenguajes de programación, la liberación de memoria previamente ocupada se maneja manualmente. Por ejemplo, en C++ se utiliza el operador **delete** para liberar memoria. Sin embargo, C# utiliza una mecánica diferente y menos compleja: la *recolección de basura*.

El sistema de recolección de basura de C# recoge objetos de manera automática, ocurre de manera transparente, tras bambalinas, sin ninguna intervención del programador. Funciona de la siguiente manera: cuando no existe ninguna referencia a un objeto, el programa supone que ese objeto ya no es necesario y la memoria que ocupa se libera en automático. Esta porción de memoria reciclada puede ser utilizada entonces por algún otro componente del programa.

La recolección de basura sólo ocurre en forma esporádica durante la ejecución de tu programa. No ocurre simplemente porque uno o más objetos existentes dejan de utilizarse. Por lo mismo, no es posible saber con precisión cuándo se producirá.

## Destructores

Es posible definir un método que será invocado justo antes de que el recolector de basura destruya finalmente un objeto. Este método recibe el nombre de *destructo*r, y puede utilizarse en situaciones altamente especializadas para asegurarse de que el objeto termine limpiamente. Por ejemplo, puedes utilizar un destructor para asegurar que el recurso del sistema utilizado por un objeto sea efectivamente liberado. Debemos dejar muy en claro que los destructores son una característica muy avanzada que se aplica sólo en ciertas situaciones especiales. Por lo regular no se les necesita. Sin embargo, como son parte de C#, hacemos una breve descripción de ellos con fines informativos.

Los destructores tienen el siguiente formato general:

```
~nombre-clase( ) {  
    // código de destrucción  
}
```

Aquí, *nombre-clase* es el nombre de la clase. Así, un destructor se declara de manera similar a un constructor, excepto que va antecedido por una tilde (~). Observa que no tiene tipo de regreso.

Es importante comprender que el destructor es invocado justo antes que el recolector de basura. No se invoca, por ejemplo, cuando una variable con una referencia a cierto objeto queda fuera de alcance. (Esto difiere de los destructores de C++, los cuales *son* invocados cuando el objeto sale de alcance.) Esto significa que tampoco puedes saber con precisión cuándo se ejecutará un destructor. Más aún, es posible que tu programa termine antes de que ocurra la recolección de basura, por lo que el destructor nunca será invocado.

## La palabra clave this

Antes de concluir este capítulo es necesario presentar una introducción sobre **this**. Cuando un método es invocado, transmite automáticamente una referencia al objeto invocador (es decir, el objeto que está llamando al método). Esta referencia recibe el nombre de **this**. Por lo mismo, **this** hace referencia al objeto sobre el que actúa el método invocado. Para comprender **this**, primero analiza el siguiente programa que crea una clase llamada **Potencia**, la cual calcula el resultado de un número elevado a una potencia entera:

```
using System;  
  
class Potencia {  
    public double b;  
    public int e;  
    public double val;  
  
    public Potencia(double num, int exp) {  
        b = num;  
        e = exp;  
  
        val = 1;  
        for( ; exp>0; exp--) val = val * b;  
    }  
}
```

```
public double ObtenerPotencia() {
    return val;
}

class DemoPotencia {
    static void Main() {
        Potencia x = new Potencia(4.0, 2);
        Potencia y = new Potencia(2.5, 1);
        Potencia z = new Potencia(5.7, 0);

        Console.WriteLine(x.b + " elevado a " + x.e + " la potencia es " +
                           x.ObtenerPotencia());
        Console.WriteLine(y.b + " elevado a " + y.e + " la potencia es " +
                           y.ObtenerPotencia());
        Console.WriteLine(z.b + " elevado a " + z.e + " la potencia es " +
                           z.ObtenerPotencia());
    }
}
```

Como sabes, dentro de un método, los demás miembros de la clase pueden ser accesados directamente, sin ninguna calificación de objeto ni clase. Así, dentro de **ObtenPotencia()**, la declaración

```
return val;
```

significa que la copia de **val** asociada con el objeto invocador será traída de regreso. Sin embargo, la misma declaración también puede ser escrita así:

```
return this.val;
```

Aquí, **this** hace referencia al objeto que está invocando el método **ObtenPotencia()**. Así, **this.val** se refiere a la copia de **val** perteneciente al objeto invocador. Por ejemplo, si **ObtenPotencia()** hubiese sido invocado en **x**, entonces el **this** del ejemplo anterior haría referencia a **x**. Escribir la declaración sin utilizar **this** es en realidad una abreviación.

También es posible utilizar **this** dentro de un constructor. En tales casos, **this** hace referencia al objeto que está siendo construido. Por ejemplo, dentro de **Potencia()**, las declaraciones

```
b = num;
e = exp;
```

pueden escribirse:

```
this.b = num;
this.e = exp;
```

Por supuesto, en este caso no se obtiene ningún beneficio al utilizar tal sintaxis.

Aquí está la clase **Potencia** completa, utilizando la referencia **this**:

```
// Muestra el uso de this.
class Potencia {
    public double b;
    public int e;
    public double val;

    public Potencia(double num, int exp) {
        this.b = num;
        this.e = exp;

        this.val = 1;
        for( ; exp>0; exp--) this.val = this.val * this.b;
    }

    public double ObtenPotencia() {
        return this.val;
    }
}
```

De hecho, ningún programador de C# utilizaría **this** para escribir **Potencia** como se acaba de mostrar, porque no se obtiene ningún beneficio y el formato estándar es más sencillo. Sin embargo, **this** tiene algunos usos importantes. Por ejemplo, C# permite que el nombre de un parámetro o de una variable local sea el mismo que una variable de instancia. Cuando esto sucede, el nombre local *oculta* la variable de instancia. Puedes obtener acceso a la variable de instancia oculta al hacer referencia a ella utilizando **this**. Por ejemplo, la siguiente es una manera sintácticamente válida para escribir el constructor **Potencia( )**:

```
public Potencia(double b, int e) {
    this.b = b; ← Aquí, this.b hace referencia a la variable de instancia b, no al parámetro.
    this.e = e;

    val = 1;
    for( ; e>0; e--) val = val * b;
}
```

En esta versión, los nombres de los parámetros son los mismos que los nombres de las variables de instancia, aunque ocultándolos. Sin embargo, **this** se utiliza para “descubrir” las variables de instancia.



### Autoexamen Capítulo 4

1. ¿Cuál es la diferencia entre una clase y un objeto?
2. ¿Cómo se define una clase?
3. ¿Qué hace que cada objeto tenga su propia copia?

- 4.** Utilizando dos declaraciones separadas, muestra cómo declarar un objeto llamado **contador** de una clase llamada **MiContador**, y asígnale una referencia hacia un objeto.
- 5.** Muestra cómo se declara un método llamado **MiMétodo( )** si tiene un tipo de regreso **double** y cuenta con dos parámetros **int** llamados **a** y **b**.
- 6.** ¿Cómo debe regresar un método si regresa un valor?
- 7.** ¿Qué nombre tiene un constructor?
- 8.** ¿Cuál es la función de **new**?
- 9.** ¿Qué es la recolección de basura y cómo funciona? ¿Qué es un destructor?
- 10.** ¿Qué es **this**?



# Capítulo 5

Más tipos de datos  
y operadores

## Habilidades y conceptos clave

- Arreglos unidimensionales
  - Arreglos multidimensionales
  - Arreglos descuadrados
  - La propiedad **Length**
  - Arreglos de tipo implícito
  - String (cadenas de caracteres)
  - El loop **foreach**
  - Los operadores bitwise
  - El operador ?
- 

Este capítulo regresa al tema de los tipos de dato y operadores de C#. Aborda los arreglos, el tipo **string**, los operadores bitwise y el operador condicional ?. A lo largo del texto, presentamos una introducción al loop **foreach**.

## Arreglos

Un *arreglo* es una colección de variables del mismo tipo a las que se hace referencia a través de un nombre común. En C#, los arreglos pueden tener una o más dimensiones, aunque los más comunes son los de una sola dimensión. Los arreglos se utilizan para una gran variedad de propósitos porque proporcionan un medio conveniente para agrupar variables relacionadas. Por ejemplo, puedes utilizar un arreglo para almacenar un registro de los días más calurosos del mes, una lista de los usuarios que ingresaron a una red o tu colección de libros de programación.

La principal ventaja de un arreglo es que organiza datos de manera que puedan ser fácilmente manipulables. Por ejemplo, si tienes un arreglo que contiene los ingresos de un grupo selecto de padres de familia, es fácil calcular el ingreso promedio haciendo un ciclo a través del arreglo. De la misma manera, los arreglos organizan datos de tal manera que pueden ser fácilmente almacenados.

Aunque los arreglos en C# pueden utilizarse como en muchos otros lenguajes de computación, tienen un atributo especial: son implementados como objetos. Por esta razón abordamos el tema de los arreglos después de haber presentado los objetos. La implementación de los arreglos como objetos implica importantes ventajas; una de ellas, de gran provecho, es que pueden ser segados por el recolector de basura.

## Arreglos unidimensionales

Un arreglo unidimensional es una lista de variables relacionadas entre sí. Estas listas son comunes en programación, por ejemplo, puedes utilizar un arreglo unidimensional para almacenar los números de cuenta de los usuarios activos de una red. Otro arreglo puede utilizarse para almacenar el promedio de bateo actual de un equipo de beisbol.

Como en C# los arreglos son implementados como objetos, es necesario realizar dos pasos para generar un arreglo que puedas utilizar en tu programa. Primero, debes declarar una variable que pueda hacer referencia a cualquier arreglo. Segundo, debes crear una instancia del arreglo utilizando la palabra clave **new**. Así, para declarar un arreglo unidimensional, por lo general utilizarás este formato estándar:

```
tipo[ ] nombre-arreglo = new tipo[tamaño];
```

Aquí, *tipo* declara el *tipo de elemento* del arreglo. El tipo de elemento determina el tipo de dato de los elementos que contiene el arreglo. Observa los corchetes que siguen al *tipo*. Indican que se está declarando una referencia a un arreglo unidimensional. La cantidad de elementos que contendrá el arreglo se determina por el elemento *tamaño*.

A continuación presentamos un ejemplo que crea un arreglo tipo **int** de diez elementos, y lo vincula a una variable de referencia del arreglo llamada **muestra**:

```
int[] muestra = new int[10];
```

La variable **muestra** contiene una referencia a la memoria reservada por **new**. Este espacio de memoria es lo suficientemente amplio para albergar diez elementos tipo **int**.

Al igual que cuando se crea una instancia de clase, es posible dividir la declaración anterior en dos partes. Por ejemplo:

```
int[] ejemplo;
muestra = new int[10];
```

En este caso, cuando **muestra** se crea por primera vez no hace referencia a ningún objeto físico. Sólo después de que se ejecuta la segunda declaración la variable **muestra** hace referencia a un arreglo.

Cada elemento individual dentro del arreglo es accesado por medio de un índice. Un *índice* describe la posición de un elemento dentro de un arreglo. En C#, todos los arreglos utilizan el cero como índice de su primer elemento. Como **muestra** tiene diez elementos, los valores de los índices van del 0 al 9. Para indexar un arreglo, especifica el número del elemento que quieras dentro de los corchetes. Así, el primer elemento en **muestra** es **muestra[0]**, y el último es **muestra[9]**. Por ejemplo, el siguiente programa carga **muestra** con números del 0 al 9:

```
// Muestra un arreglo unidimensional.

using System;

class ArregloDemo {
    static void Main() {
        int[] muestra = new int[10];
        int i;
```

```
for(i = 0; i < 10; i++) ←  
    muestra[i] = i;  
  
for(i = 0; i < 10; i++) ←  
    Console.WriteLine("Ésta es una muestra[" + i + "]: " + muestra[i]);  
}  
}
```

Los índices de los arreglos comienzan en cero.

Los datos de salida del programa son los siguientes:

```
Ésta es una muestra[0]: 0  
Ésta es una muestra[1]: 1  
Ésta es una muestra[2]: 2  
Ésta es una muestra[3]: 3  
Ésta es una muestra[4]: 4  
Ésta es una muestra[5]: 5  
Ésta es una muestra[6]: 6  
Ésta es una muestra[7]: 7  
Ésta es una muestra[8]: 8  
Ésta es una muestra[9]: 9
```

Conceptualmente, el arreglo **muestra** se ve así:

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

Los arreglos son comunes en programación porque te permiten manejar fácilmente grandes cantidades de variables relacionadas. Por ejemplo, el siguiente programa encuentra los valores mínimo y máximo dentro del arreglo **nums**, realizando ciclos reiterativos con el loop **for**:

```
// Encuentra los valores mínimo y máximo dentro del arreglo.  
  
using System;  
  
class MinMax {  
    static void Main() {  
        int[] nums = new int[10];  
        int min, max;  
  
        nums[0] = 99;  
        nums[1] = -10;  
        nums[2] = 100123;  
        nums[3] = 18;
```

```

nums[4] = -978;
nums[5] = 5623;
nums[6] = 463;
nums[7] = -9;
nums[8] = 287;
nums[9] = 49;

min = max = nums[0];

// Encuentra los valores mínimo y máximo dentro del arreglo.
for(int i=1; i < 10; i++) {
    if(nums[i] < min) min = nums[i];
    if(nums[i] > max) max = nums[i];
}

Console.WriteLine("mínimo y máximo: " + min + " " + max);
}
}

```

Los datos de salida que presenta el programa son:

```
mínimo y máximo: -978 100123
```

### Inicializar un arreglo

En el programa anterior, al arreglo **nums** se le insertaron los valores manualmente, utilizando declaraciones de asignación por separado. Esto es perfectamente correcto, pero existe un medio más fácil de realizar la tarea. Los arreglos pueden ser inicializados cuando se crean. El formato general para inicializar un arreglo unidimensional se muestra a continuación:

```
tipo[ ] nombre-arreglo = { valor1, valor2, valor3, ..., valorN };
```

En este caso, los valores iniciales se especifican desde *valor1* hasta *valorN*. Están asignados en secuencia, de izquierda a derecha, en orden de indexado. C# reserva de manera automática una porción de memoria lo bastante grande para contener los inicializadores que especificaste. No es necesario hacer uso explícito del operador **new**. Por ejemplo, a continuación presentamos una mejor manera de escribir el programa **MinMax**:

```

// Utilizar inicializadores de arreglo.

using System;

class MinMax {
    static void Main() {
        int[] nums = { 99, -10, 100123, 18, -978,
                      5623, 463, -9, 287, 49 }; ← Inicializadores del arreglo.
        int min, max;
    }
}

```

```
min = max = nums[0];
for(int i=1; i < 10; i++) {
    if(nums[i] < min) min = nums[i];
    if(nums[i] > max) max = nums[i];
}
Console.WriteLine("mínimo y máximo: " + min + " " + max);
}
```

Como punto de interés, aunque no es necesario, puedes utilizar el operador **new** cuando inicializas un arreglo. Por ejemplo, la siguiente es una manera apropiada, aunque redundante, de inicializar **nums** del programa anterior:

```
int[] nums = new int[] { 99, -10, 100123, 18, -978,
                        5623, 463, -9, 287, 49 };
```

Aunque en el caso anterior es redundante, el uso del operador **new** es útil cuando asignas un nuevo arreglo a una variable de referencia a un arreglo ya existente. Por ejemplo:

```
int[] nums;
nums = new int[] { 99, -10, 100123, 18, -978,
                  5623, 463, -9, 287, 49 };
```

En este caso, **nums** es declarado en la primera declaración e inicializado en la segunda.

### Los límites son obligatorios

Los límites de un arreglo son estrictamente obligatorios en C#: exceder o reducir los límites del arreglo traerá como resultado un error en tiempo de ejecución. Si quieres confirmarlo por ti mismo, intenta ejecutar el siguiente programa que excede a propósito el límite de un arreglo:

```
// Muestra el exceso de un arreglo.

using System;

class ArrayErr {
    static void Main() {
        int[] ejemplo = new int[10];

        // Genera un desbordamiento del arreglo.
        for(int i = 0; i < 100; i++)
            ejemplo[i] = i;
    }
}
```

↑  
Excede el límite de **muestra**.

En cuanto **i** alcanza 10 se genera un error **IndexOutOfRangeException** y el programa se aborta.

**Prueba esto****Organizar un arreglo**

Como un arreglo unidimensional organiza datos en una lista lineal indizable, representa una estructura perfecta para ordenarlos. Más aún, los algoritmos de organización son los mejores ejemplos para el uso de arreglos. Siguiendo ese fin, el siguiente ejemplo desarrolla una versión de la poco apreciada *organización bubble* (burbuja). Como tal vez lo sepas, existe una gran variedad de algoritmos para organizar datos. Entre ellos se cuentan la organización rápida, la organización aleatoria y tipo Shell, por mencionar sólo tres de ellas. Sin embargo, uno de los algoritmos de organización más sencillos y fáciles de comprender es la bubble. También es uno de los peores en términos de rendimiento (aunque en ocasiones puede ser efectivo para ordenar arreglos pequeños). No obstante, la organización bubble tiene una característica útil: ¡es un ejemplo interesante para el manejo de arreglos!

**Paso a paso**

- 1.** Crea un archivo llamado **Bubble.cs**.
- 2.** El arreglo bubble recibe ese nombre por la manera de realizar la operación organizativa. Utiliza comparaciones repetitivas y, de ser necesario, intercambia los elementos adyacentes dentro del arreglo. En este proceso, los valores pequeños se trasladan a un extremo del arreglo y los grandes al otro. El arreglo es conceptualmente similar a la manera en que las burbujas encuentran su propio nivel en un tanque de agua. La organización bubble opera realizando varios pasos a través del arreglo y moviendo elementos de lugar cuando es necesario. La cantidad de pasos necesarios para asegurar que el arreglo está finalmente organizado es igual a la cantidad de elementos dentro del arreglo menos uno.

He aquí el código elemental del arreglo bubble. El arreglo que se organiza en este ejemplo recibe el nombre de **nums**.

```
// Ésta es la organización bubble.
for(a=1; a < tamaño; a++) {
    for(b=tamaño-1; b >= a; b--) {
        if(nums[b-1] > nums[b]) {
            // Intercambia los elementos organizados.
            t = nums[b-1];
            nums[b-1] = nums[b];
            nums[b] = t;
        }
    }
}
```

Advierte que la organización depende de dos loops **for**. El loop interno verifica los elementos adyacentes del arreglo en busca de los que están fuera de su lugar. Cuando encuentra una pareja en tales condiciones, los intercambia de lugar. Con cada pase, los elementos más pequeños de los restantes son movidos y colocados en el lugar que les corresponde. El loop exterior hace que el proceso se repita hasta que el arreglo completo ha sido organizado.

- 3.** He aquí el programa **Bubble** completo:

```
// Muestra la organización bubble.
using System;
```

(continúa)

```
class Bubble {
    static void Main() {
        int[] nums = { 99, -10, 100123, 18, -978,
                      5623, 463, -9, 287, 49 };
        int a, b, t;
        int tamaño;

        // Establece la cantidad de elementos a organizar.
        tamaño = 10;

        // Muestra el arreglo original.
        Console.Write("El arreglo original es:");
        for(int i=0; i < tamaño; i++)
            Console.Write(" " + nums[i]);
        Console.WriteLine();

        // Ésta es la organización bubble.
        for(a=1; a < tamaño; a++) {
            for(b=tamaño-1; b >= a; b--) {
                if(nums[b-1] > nums[b]) {
                    // Intercambia los elementos organizados.
                    t = nums[b-1];
                    nums[b-1] = nums[b];
                    nums[b] = t;
                }
            }
        }

        // Muestra el arreglo organizado.
        Console.Write("El arreglo organizado es:");
        for(int i=0; i < tamaño; i++)
            Console.Write(" " + nums[i]);
        Console.WriteLine();
    }
}
```

Los datos generados por el programa son los siguientes:

```
El arreglo original es: 99 -10 100123 18 -978 5623 463 -9 287 49
El arreglo organizado es: -978 -10 -9 18 49 99 287 463 5623 100123
```

4. Como se muestra en el ejemplo, **Bubble** organiza un arreglo de datos tipo **int**, pero puedes cambiar el tipo de dato de los elementos que serán organizados, cambiando el tipo del arreglo **nums** y el tipo de la variable **t**. Para organizar, por ejemplo, un arreglo de números de punto flotante, tanto **nums** como **t** deben ser de tipo **double**.
5. Como se mencionó, aunque la organización bubble proporciona un excelente ejemplo para el manejo de arreglos, en la mayoría de los casos no representa una opción viable. El mejor algoritmo de propósito general para la clasificación de datos es la organización rápida. Pero ésta utiliza características de C# que se abordan en el capítulo 6, por lo que pospondremos hasta entonces su explicación.

## Arreglos multidimensionales

Aunque el arreglo unidimensional es el más utilizado, los multidimensionales no son raros en programación. Un arreglo multidimensional tiene dos o más dimensiones y sus elementos individuales son accesados por la combinación de dos o más índices.

### Arreglos bidimensionales

La forma más sencilla del arreglo multidimensional es el de dos dimensiones o bidimensional. En éste, la localización de un elemento se especifica por dos índices. Piensa en los arreglos bidimensionales como una tabla de información: un índice indica la línea y el otro la columna.

Para declarar un arreglo bidimensional de enteros llamado **tabla**, con un tamaño 10, 20, debes escribir

```
int[,] tabla = new int[10, 20];
```

Presta mucha atención a esta declaración. Advierte que las dos dimensiones están separadas por una coma. En la primera parte de la declaración, la sintaxis

```
[ , ]
```

indica que se está creando una variable de referencia de un arreglo bidimensional. Cuando la memoria es de hecho reservada para el arreglo utilizando el operador **new**, se utiliza esta sintaxis:

```
int[10, 20];
```

Esto crea un arreglo de 10x20, y de nuevo, la coma separa las dimensiones.

Para accesar un elemento dentro de un arreglo bidimensional, debes especificar los dos índices, separando uno de otro con una coma. Por ejemplo, para asignar el valor 10 al lugar 3, 5 del arreglo **tabla**, utilizarías la siguiente instrucción:

```
tabla[3, 5] = 10;
```

A continuación presentamos un ejemplo completo. Carga un arreglo bidimensional con los números del 1 al 12 y muestra el contenido del arreglo.

```
// Muestra un arreglo bidimensional.

using System;

class DosD {
    static void Main() {
        int t, i;
        int[,] tabla = new int[3, 4]; ← Declarar un arreglo bidimensional de 3 por 4.

        for(t=0; t < 3; ++t) {
            for(i=0; i < 4; ++i) {
                tabla[t,i] = (t*4)+i+1;
                Console.Write(tabla[t,i] + " ");
            }
        }
    }
}
```

```
        Console.WriteLine();  
    }  
}
```

En este ejemplo, **tabla[0, 0]** tendrá el valor 1, **tabla[0, 1]** el valor 2, **tabla[0, 2]** el valor 3, y así sucesivamente. El valor de **tabla[2, 3]** será 12. Conceptualmente, el arreglo tendrá la apariencia presentada en la figura 5-1.

## Arreglos de tres o más dimensiones

C# permite la existencia de arreglos con más de dos dimensiones. He aquí el formato general de una declaración de arreglo multidimensional:

*tipo[,...,] nombre = new tipo[tamaño1, tamaño2,...,tamañoN];*

Por ejemplo, la siguiente declaración crea un arreglo cúbico de enteros  $4 \times 10 \times 3$ :

```
int[, , ] multidim = new int[4, 10, 3];
```

Para asignar el valor de 100 al elemento 2, 4, 1 de **multidim**, utiliza la siguiente declaración:

```
multidim[2, 4, 1] = 100;
```

# Inicializar arreglos multidimensionales

Un arreglo multidimensional puede ser inicializado encerrando los datos de cada dimensión dentro de su propio juego de llaves. Por ejemplo, el formato general para inicializar un arreglo bidimensional es el siguiente:

```

tipo[nombre_arreglo]= {
    {val, val, val, ..., val },
    {val, val, val, ..., val },
    .
    .
    .
    {val, val, val, ..., val }
};
```

	0	1	2	3	índice derecho
0	1	2	3	4	
1	5	6	(7)	8	
2	9	10	11	12	

índice izquierdo

tabla[1][2]

**Figura 5-1** Una vista conceptual del arreglo **tabla**, creado por el programa **DosD**

Aquí, *val* indica el valor de inicio. Cada bloque interno designa una fila. Dentro de cada fila el primer valor será almacenado en la primera celda de la misma, el segundo valor en la segunda y así sucesivamente. Observa que los bloques inicializadores están separados por comas, y que después de la llave de clausura se escribe un punto y coma.

Por ejemplo, el siguiente programa inicializa un arreglo llamado **cuadrados** con los números 1 a 5 y sus respectivos cuadrados:

```
// Inicializa un arreglo bidimensional.

using System;

class Cuadrados {
    static void Main() {
        int[,] cuadrados = {
            { 1, 1 },
            { 2, 4 },
            { 3, 9 },
            { 4, 16 },
            { 5, 25 },
        };
        int i, j;

        for(i=0; i < 5; i++) {
            for(j=0; j < 2; j++)
                Console.Write(cuadrados[i,j] + " ");
            Console.WriteLine();
        }
    }
}
```

← Observa cómo cada fila tiene sus propios inicializadores.

El programa genera los siguientes datos:

```
1 1
2 4
3 9
4 16
5 25
```

## Arreglos descuadrados

En los ejemplos anteriores, cuando creaste arreglos bidimensionales, creaste lo que en C# se llaman *arreglos rectangulares*. Pensando en los arreglos bidimensionales como tablas, un arreglo rectangular es uno bidimensional donde el largo de cada fila es siempre el mismo. No obstante, C# también te permite crear un tipo especial de arreglo bidimensional llamado *arreglo descuadrado*, el cual es un *arreglo de arreglos* donde la longitud de cada uno de ellos puede ser diferente del resto. Así, un arreglo descuadrado puede ser utilizado para crear una tabla donde las longitudes de las filas sean diferentes unas de otras.

Los arreglos descuadrados pueden ser declarados utilizando conjuntos de corchetes para indicar cada dimensión. Por ejemplo, para declarar un arreglo bidimensional descuadrado, utilizarás el siguiente formato general:

```
tipo[ ][ ] nombre-arreglo = new tipo[tamaño][ ];
```

Aquí, *tamaño* indica la cantidad de filas en el arreglo. Las filas en sí no han sido reservadas en memoria. En lugar de ello, las filas son reservadas de manera individual. Esto es lo que permite que cada una de ellas tenga una longitud diferente. Por ejemplo, el siguiente código reserva memoria para la primera dimensión **descuadrada** cuando se declara. Después reserva la segunda dimensión manualmente.

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

Después de que se ejecuta la anterior secuencia, el arreglo descuadrado queda así:

descua [0][0]	descua [0][1]		
descua [1][0]	descua [1][1]	descua [1][2]	
descua [2][0]	descua [2][1]	descua [2][2]	descua [2][3]

¡Es fácil saber de dónde viene el nombre de este tipo de arreglo!

Una vez que se ha creado el arreglo descuadrado, un elemento se accesa especificando cada índice dentro de su propio juego de corchetes. Por ejemplo, para asignar el valor 10 al elemento 2, 1 de **descua**, se utilizaría la siguiente declaración:

```
descua[2][1] = 10;
```

Observa que esta sintaxis es diferente a la utilizada para accesar un elemento en un arreglo rectangular.

A continuación presentamos un ejemplo que utiliza un arreglo descuadrado bidimensional. Supongamos que estás escribiendo un programa que almacena la cantidad de pasajeros que se transportan en el transbordador del aeropuerto. Si el transbordador corre diez veces al día durante los días hábiles y dos veces al día los sábados y domingos, puedes utilizar el arreglo **pasajeros** que se presenta en el siguiente programa para almacenar la información. Advierte que la longitud de la segunda dimensión para las primeras cinco es 10 y la longitud de la segunda dimensión para las últimas dos es 2.

```
// Muestra arreglos descuadrados.  
  
using System;  
  
class Descuadrado {  
    static void Main() {  
        int[][] pasajeros = new int[7][];
```

```
pasajeros[0] = new int[10];
pasajeros[1] = new int[10];
pasajeros[2] = new int[10]; ← Aquí, las segundas dimensiones tienen
pasajeros[3] = new int[10];
pasajeros[4] = new int[10];

pasajeros[5] = new int[2]; ← Pero aquí, sólo tienen 2 elementos de longitud.
pasajeros[6] = new int[2];

int i, j;

// Fabrica algunos datos.
for(i=0; i < 5; i++)
    for(j=0; j < 10; j++)
        pasajeros[i][j] = i + j + 10;
for(i=5; i < 7; i++)
    for(j=0; j < 2; j++)
        pasajeros[i][j] = i + j + 10;

Console.WriteLine("Pasajeros por viaje durante la semana:");
for(i=0; i < 5; i++) {
    for(j=0; j < 10; j++)
        Console.Write(pasajeros[i][j] + " ");
    Console.WriteLine();
}
Console.WriteLine();

Console.WriteLine("Pasajeros por viaje los fines de semana:");
for(i=5; i < 7; i++) {
    for(j=0; j < 2; j++)
        Console.Write(pasajeros[i][j] + " ");
    Console.WriteLine();
}
}
```

Los arreglos descuadrados no se utilizarán en todas las aplicaciones, pero pueden ser muy efectivos en ciertas situaciones. Por ejemplo, si necesitas utilizar un arreglo bidimensional muy grande con datos esparcidos (es decir, donde no todos los elementos serán utilizados), el arreglo descuadrado puede ser una solución perfecta.

## Asignar referencias a arreglos

Como con otros objetos, cuando asignas una variable de referencia de un arreglo hacia otro, simplemente estás haciendo que ambas variables hagan referencia al mismo arreglo. No estás pro-

vocando que se haga una copia del arreglo, ni que el contenido de un arreglo se copie a otro. Por ejemplo, analiza el siguiente programa:

```
// Asignar variables de referencia a arreglos.

using System;

class AsignaRef {
    static void Main() {
        int i;

        int[] nums1 = new int[10];
        int[] nums2 = new int[10];

        // Inserta algunos valores a nums1 y nums2.
        for(i=0; i < 10; i++) nums1[i] = i;
        for(i=0; i < 10; i++) nums2[i] = -i;

        Console.WriteLine("Aquí está nums1: ");
        for(i=0; i < 10; i++)
            Console.Write(nums1[i] + " ");
        Console.WriteLine();

        Console.WriteLine("Aquí está nums2: ");
        for(i=0; i < 10; i++)
            Console.Write(nums2[i] + " ");
        Console.WriteLine();

        // Ahora nums2 hace referencia a nums1.
        nums2 = nums1; ← Asigna una referencia de arreglo a otro.

        Console.WriteLine("Aquí está nums2 después de la asignación: ");
        for(i=0; i < 10; i++)
            Console.Write(nums2[i] + " ");
        Console.WriteLine();

        // Opera sobre nums1 a través de nums2.
        nums2[3] = 99;

        Console.WriteLine("Aquí está nums1 después de cambiar a través de
nums2: ");
        for(i=0; i < 10; i++)
            Console.Write(nums1[i] + " ");
        Console.WriteLine();
    }
}
```

El programa genera los siguientes datos de salida:

```
Aquí está nums1: 0 1 2 3 4 5 6 7 8 9
Aquí está nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Aquí está nums2 después de la asignación: 0 1 2 3 4 5 6 7 8 9
Aquí está nums1 después de cambiar a través de nums2: 0 1 2 99 4 5 6 7
                                                               8 9
```

Como lo muestran los datos de salida, después de asignar **nums1** a **nums2**, ambas variables hacen referencia al mismo objeto.

## Utilizar la propiedad de longitud con arreglos

Implementar arreglos como objetos en C# trajo una gran cantidad de beneficios. Uno de ellos proviene del hecho de que cada arreglo está asociado con una propiedad **Length** (longitud), que contiene la cantidad de elementos que un arreglo puede contener. Así, cada arreglo proporciona un medio a través del cual su longitud puede ser determinada. A continuación presentamos un programa que muestra la aplicación de la propiedad **Length**:

```
// Utilización de la propiedad Length de un arreglo.

using System;

class LengthDemo {
    static void Main() {
        int[] lista = new int[10];
        int[,] dosD = new int[3, 4];
        int[] nums = { 1, 2, 3 };

        // Una tabla de longitud variable.
        int[][] tabla = new int[3][];

        // Añade segundas dimensiones.
        tabla[0] = new int[] { 1, 2, 3 };
        tabla[1] = new int[] { 4, 5 };
        tabla[2] = new int[] { 6, 7, 8, 9 };

        Console.WriteLine("la longitud de lista es " + lista.Length);
        Console.WriteLine("la longitud de dosD es " + dosD.Length);
        Console.WriteLine("la longitud de nums es " + nums.Length);
        Console.WriteLine("la longitud de tabla es " + tabla.Length);
        Console.WriteLine("la longitud de tabla[0] es " + tabla[0].Length);
        Console.WriteLine("la longitud de tabla[1] es " + tabla[1].Length);
        Console.WriteLine("la longitud de tabla[2] es " + tabla[2].Length);
        Console.WriteLine();

        // Utiliza length para inicializar la lista.
        for(int i=0; i < lista.Length; i++) ← Usa Length para controlar el loop for.
            lista[i] = i * i;
    }
}
```

Muestra la longitud de los arreglos.

```
Console.WriteLine("He aquí la lista: ");
// Ahora usa length para mostrar la lista
for(int i=0; i < lista.Length; i++)
    Console.Write(lista[i] + " ");
Console.WriteLine();
}
}
```

El programa genera los siguientes datos de salida:

```
la longitud de lista es 10
la longitud de dosD es 12
la longitud de nums es 3
la longitud de tabla es 3
la longitud de tabla[0] es 3
la longitud de tabla[1] es 2
la longitud de tabla[2] es 4

He aquí la lista: 0 1 4 9 16 25 36 49 64 81
```

Hay muchos puntos de interés en este programa. Primero, la longitud de **lista**, que es un arreglo unidimensional, es igual a su declaración de 10. Segundo, la longitud de **dosD**, un arreglo bidimensional de 3 por 4, es 12, que es la cantidad total de elementos que contiene. En general, la longitud de un arreglo multidimensional es igual a todos los elementos que puede contener. Sin embargo, la situación es diferente para los arreglos descuadrados.

Como sabes, el descuadrado es un arreglo de arreglos. Por tanto, en el programa **tabla** es un arreglo descuadrado de dos dimensiones. Presta especial atención a la manera como se utiliza **Length** dentro de él. Primero, la expresión

```
tabla.Length
```

obtiene la cantidad de *arreglos* almacenados en **tabla**, que en este caso es 3. Para conocer la longitud de cualquier arreglo individual dentro de **tabla**, se utiliza una expresión como la siguiente:

```
tabla[0].Length
```

con esta expresión en particular se determina la longitud del primer arreglo, que es igual a 3.

Otro aspecto de importancia que debemos resaltar del programa **LengthDemo** es la manera en que los loops **for** utilizan la expresión **lista.Length** para controlar la cantidad de repeticiones que se realizan. Como cada arreglo tiene su propia longitud, puedes utilizar esta información en lugar de rastrear manualmente el tamaño del arreglo. Ten presente que el valor de **Length** no tiene relación alguna con la cantidad de elementos que en realidad estás utilizando. Contiene la cantidad de elementos que el arreglo es capaz de contener.

La inclusión de la propiedad **Length** simplifica muchos algoritmos, al facilitar cierto tipo de operaciones en los arreglos y hacerlos más seguros. Por ejemplo, el siguiente programa utiliza

**Length** para copiar un arreglo en otro, previniendo que se desborde y aparezca una excepción en tiempo de ejecución:

```
// Utiliza la propiedad Length como ayuda para copiar un arreglo.

using System;

class UnaCopia {
    static void Main() {
        int i;
        int[] nums1 = new int[10];
        int[] nums2 = new int[10];

        for(i=0; i < nums1.Length; i++) nums1[i] = i;

        // Copia nums1 a nums2.
        if(nums2.Length >= nums1.Length) ←
            for(i = 0; i < nums2.Length; i++)
                nums2[i] = nums1[i];

        for(i=0; i < nums2.Length; i++)
            Console.Write(nums2[i] + " ");
    }
}
```

Confirma que el arreglo blanco es lo bastante grande para contener todos los elementos del arreglo fuente.

Aquí, **Length** ayuda a realizar dos importantes funciones. Primero, es usado para confirmar que el arreglo destino es lo suficientemente grande para contener los datos del arreglo fuente. Segundo, proporciona la condición terminal del loop **for** que realiza la copia. Por supuesto, se trata de un ejemplo sencillo donde la longitud de los arreglos se puede conocer fácilmente, pero el mismo principio puede aplicarse a un amplio rango de situaciones más complejas.

## Crear un tipo de arreglo implícito

Como se explicó en el capítulo 2, C# 3.0 añade la capacidad de declarar variables de tipo implícito utilizando la palabra clave **var**. Éstas son variables cuyo tipo es determinado por el compilador, con base en el tipo de la expresión de inicio. De esta manera, todas las variables de tipo implícito deben ser inicializadas. Utilizando el mismo mecanismo, también es posible crear un arreglo de tipo implícito. Como regla general, los arreglos de tipo implícito se utilizan en cierto tipo de búsquedas en una base de datos (queries) relacionados con LINQ, los cuales se estudian más adelante en este mismo libro. En la mayoría de los casos utilizarás el mecanismo de declaración “normal” en los arreglos. Los arreglos de tipo implícito son presentados en este punto con el fin de completar el tema.

Un arreglo de tipo implícito se declara utilizando la palabra clave **var**, pero después de ella *no se colocan* los corchetes ([ ]). Más aún, el arreglo debe ser inicializado. Es la clase de inicialización que determina el tipo de los elementos que contiene el arreglo. Todos los inicializadores deben ser del mismo tipo o de uno compatible. He aquí un ejemplo de un arreglo de tipo implícito:

```
var vals = new[] { 1, 2, 3, 4, 5 };
```

Esto crea un arreglo de enteros (**int**) con una longitud de cinco elementos. Una referencia a ese arreglo se le asigna a **vals**. Así, el tipo **vals** es un “arreglo de **int**” y tiene cinco elementos. De nuevo advierte que **var** no está seguida por los corchetes **[ ]**. De igual modo, aunque el arreglo ha sido iniciado, de cualquier manera debes incluir la palabra clave **new[ ]**. No es opcional en este contexto.

He aquí otro ejemplo. Éste crea un arreglo bidimensional de tipo **double**.

```
var vals = new[,] { {1.1, 2.2}, {3.3, 4.4}, {5.5, 6.6} };
```

En este caso, **vals** tiene dimensiones de 2 por 3.

Como se mencionó, los arreglos de tipo implícito son más aplicables a queries basados en LINQ. No están planeados para uso general. En la mayoría de los casos deberás utilizar arreglos de tipo explícito.

## Prueba esto

## Crear una clase sencilla de estructura en cola

Como tal vez ya lo sepas, una *estructura de datos* es un medio para organizar datos. La estructura de datos más sencilla es el arreglo, que es una lista lineal que soporta acceso aleatorio a sus elementos. Por lo general, los arreglos son utilizados como la base para estructuras de datos más sofisticadas, como en pila y en cola. Una *pila* es una lista en la cual los elementos sólo pueden ser accesados en el orden “el primero que entra es el último que sale” (FILO por sus siglas en inglés). Una *cola* es una lista en la cual los elementos sólo pueden ser accesados en el orden “el primero que entra es el primero que sale” (FIFO, por sus siglas en inglés). Así, las pilas son como las pilas de platos en una mesa: el primero es puesto hasta abajo y es el último que se ocupa. La cola, por su parte, es como la fila del banco: el primero que llega es el primero en ser atendido.

Lo que hace interesantes las estructuras de datos como las pilas y las colas es que combinan la información de almacenamiento con los métodos que accesan esa misma información. De esta manera, las pilas y las colas son *motores de datos* en donde el almacenamiento y la recuperación de información son proporcionados por la misma estructura de datos, no es una acción que se tenga que programar en forma manual en la aplicación. Tal combinación es, obviamente, una opción excelente para una clase, y en este ejercicio crearás una sencilla clase de estructura de datos en cola.

En general, las colas soportan dos operadores básicos: *put* y *get*. Cada operación *put* coloca un nuevo elemento al final de la cola. Cada operación *get* trae el siguiente elemento que se encuentra al inicio de la cola. Las operaciones de cola son extintivas: una vez que el elemento ha sido recuperado, no puede traerse de nuevo. La cola también puede llenarse si no hay más espacio disponible para almacenar otro elemento y puede vaciarse si se han recuperado todos los elementos.

Existen muchas y muy variadas maneras de implementar una cola. Para ser breves, la que implementaremos en este ejemplo es quizá la más sencilla. Crea una cola de uso individual y tamaño fijo, en la cual los lugares vacíos *no son* reutilizables. De esta manera, durante el proceso la cola se extingue y es descartada. Aunque la funcionalidad de este tipo de colas es muy limitada, en ocasiones es muy útil. Por ejemplo, una cola de ese tipo puede contener una lista de datos producidos por un proceso y consumidos por otro. En tal situación, ni el productor ni el consumidor querrán reutilizar la cola. Cuando se producen más datos, se crea una nueva cola, la anterior simplemente es descartada y eliminada en su momento por el recolector de basura.

Por supuesto, es posible programar colas mucho más sofisticadas. Más adelante en este mismo libro verás una manera diferente de implementar una cola, llamada *cola circular*, que reutiliza los espacios vacíos del arreglo que la alimenta cuando un elemento es recuperado. De esta manera, una cola circular puede seguir colocando elementos siempre y cuando otros sean recuperados. También aprenderás a crear colas dinámicas, que se expanden automáticamente para almacenar más elementos. Pero por el momento, una cola sencilla será suficiente.

## Paso a paso

**1.** Crea un archivo llamado **QDemo.cs**.

**2.** Aunque existen otras maneras de soportar una cola, el método que utilizaremos se basa en un arreglo. Es decir, en el arreglo se almacenarán los elementos que harán la cola. Este arreglo será accesado a través de dos índices. El índice *put* determina dónde será almacenado el siguiente dato. El índice *get* indica de qué lugar será extraído el siguiente dato. Ten presente que la operación *get* es extintiva y que no es posible extraer el mismo elemento dos veces. Aunque la cola que vamos a crear almacena caracteres, la misma lógica se puede utilizar para almacenar cualquier otro tipo de objetos. Comienza creando la clase **ColaSimple** con estas líneas:

```
class ColaSimple {
    public char[] q; // este arreglo contiene la cola
    public int putloc, getloc; // los índices put y get
```

**3.** El constructor para la clase **ColaSimple** crea la cola y le asigna tamaño. Aquí está el constructor de **ColaSimple**:

```
public ColaSimple(int tamaño) {
    q = new char[tamaño+1]; // reserva memoria para la cola
    putloc = getloc = 0;
}
```

Observa que la longitud de la cola es creada con un elemento más que el especificado en **tamaño**. Esto se debe a la manera como será implementado el algoritmo de la cola: un lugar del arreglo quedará sin ser utilizado, por lo que debe tener un espacio más al requerido por el tamaño completo de la cola. Los índices **putloc** y **getloc** inician en cero.

**4.** El método **Put( )**, que almacena elementos, se muestra a continuación:

```
// Coloca un carácter en cola.
public void Put(char ch) {
    if(putloc==q.Length-1) {
        Console.WriteLine(" -- La cola está llena.");
        return;
    }
    putloc++;
    q[putloc] = ch;
}
```

El método comienza verificando que la cola no esté llena. Si **putloc** es igual al último lugar en el arreglo **q**, ya no hay lugar para almacenar más elementos. En caso contrario, **putloc** se in-

(continúa)

crementa y el nuevo elemento es almacenado en ese lugar. Así, **putloc** es siempre el índice del último elemento almacenado.

5. Para recuperar elementos, utiliza el método **Get()**, como se muestra a continuación:

```
// Recupera un carácter de la cola.
public char Get() {
    if(getloc == putloc) {
        Console.WriteLine(" -- La cola está vacía.");
        return (char) 0;
    }
    getloc++;
    return q[getloc];
}
```

Observa que primero verifica si la cola está vacía. Si los índices **getloc** y **putloc** tienen el mismo elemento, entonces se entiende que la cola está vacía. Por esa razón, tanto **getloc** como **putloc** son inicializados en cero por el constructor **ColaSimple**. A continuación, **getloc** se incrementa y el siguiente elemento es recuperado. Así, **getloc** siempre indica la última localización del último elemento recuperado.

6. A continuación presentamos el programa **QDemo.cs** completo:

```
// Una clase de cola sencilla para caracteres.
using System;

class ColaSimple {
    public char[] q; // este arreglo contiene la cola
    public int putloc, getloc; // los índices put y get

    public ColaSimple(int tamaño) {
        q = new char[tamaño+1]; // reserva memoria para la cola
        putloc = getloc = 0;
    }

    // Coloca un carácter en cola.
    public void Put(char ch) {
        if(putloc==q.Length-1) {
            Console.WriteLine(" -- La cola está llena.");
            return;
        }
        putloc++;
        q[putloc] = ch;
    }

    // Recupera un carácter de la cola.
    public char Get() {
        if(getloc == putloc) {
            Console.WriteLine(" -- La cola está vacía.");
            return (char)0;
        }
    }
}
```

```
        getloc++;
        return q[getloc];
    }
}

// Muestra la clase de cola sencilla.
class QDemo {
    static void Main() {
        ColaSimple ColaG = new ColaSimple(100);
        ColaSimple ColaP = new ColaSimple(4);
        char ch;
        int i;

        Console.WriteLine("Usa colag para almacenar el alfabeto.");
        // pone algunos números en colaG
        for(i=0; i < 26; i++)
            ColaG.Put((char)('A' + i));

        // Recupera y muestra elementos de colag.
        Console.Write("Contenido de colag: ");
        for(i=0; i < 26; i++) {
            ch = ColaG.Get();
            if(ch != (char) 0) Console.Write(ch);
        }

        Console.WriteLine("\n");

        Console.WriteLine("Usa colap para generar errores.");
        // Ahora utiliza colap para generar algunos errores.
        for(i=0; i < 5; i++) {
            Console.Write("Intenta almacenar " +
                (char) ('Z' - i));

            ColaP.Put((char) ('Z' - i));

            Console.WriteLine();
        }
        Console.WriteLine();

        // Más errores en colap.
        Console.Write("Contenido de colap: ");
        for(i=0; i < 5; i++) {
            ch = ColaP.Get();

            if(ch != (char) 0) Console.Write(ch);
        }
    }
}
```

7. Los datos de salida generados por el programa son:

Usa ColaG para almacenar el alfabeto.

Contenido de ColaG: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Usa ColaP para generar errores.

Intenta almacenar Z

Intenta almacenar Y

Intenta almacenar X

Intenta almacenar W

Intenta almacenar V -- La cola está llena.

Contenido de ColaP: ZYXW -- La cola está vacía.

8. Por cuenta propia intenta modificar **ColaSimple** de manera que almacene tipos diferentes de objetos. Por ejemplo, haz que almacene valores **int** o **double**.

---

## El loop **foreach**

En el capítulo 3 se mencionó que C# define un loop llamado **foreach**, pero que su explicación se posponía hasta que supieras un poco más sobre el lenguaje. El momento de la explicación ha llegado.

El loop **foreach** es utilizado para hacer repeticiones cíclicas a través de los elementos de una *colección*. Una colección es un grupo de objetos. C# define varios tipos de colecciones, una de las cuales es el arreglo. El formato general de **foreach** es el siguiente:

*foreach(tipo loopvar in colección) declaración;*

Aquí, *tipo loopvar* especifica el tipo y el nombre de una *variable de reiteración*. La variable de reiteración recibe el valor del siguiente elemento de la colección cada vez que se repite el loop **foreach**. La colección sobre la que se hacen las repeticiones cíclicas se especifica en *colección*, la cual, durante el resto de la explicación, será considerada un arreglo. De esta manera, el *tipo* de la variable de reiteración debe ser el mismo, o por lo menos compatible, con el tipo de los elementos de la colección. A partir de la versión 3.0 de C#, el *tipo* de la variable de reiteración también puede ser **var**, en cuyo caso el compilador lo determina basándose en el tipo de los elementos del arreglo. Esto puede ser útil cuando se trabaja con ciertas instrucciones de bases de datos (queries), como se explicará más tarde en este mismo libro. Por lo regular, tendrás que especificar explícitamente el tipo de los datos.

Veamos ahora cómo funciona el loop **foreach**. Cuando inicia el loop, el primer elemento del arreglo es obtenido y asignado a *loopvar*. Cada repetición subsiguiente obtiene el siguiente elemento del arreglo y lo almacena en la misma variable. El loop finaliza cuando ya no hay más elementos por obtener. De esta manera, **foreach** realiza los ciclos repetitivos sobre el arreglo un elemento a la vez, de principio a fin.

Un punto importante a recordar sobre **foreach** es que la variable de reiteración *loopvar* es de solamente lectura. Esto significa que no puedes cambiar el contenido de un arreglo asignándole un nuevo valor a esta variable.

A continuación presentamos un ejemplo que utiliza **foreach**. Crea un arreglo de enteros y le da algunos valores iniciales. Después utiliza el loop **foreach** para mostrar esos valores, calculando la sumatoria de ellos en el proceso.

```
// Uso de foreach

using System;

class ForeachDemo {
    static void Main() {
        int sum = 0;
        int[] nums = new int[10];

        // Da a nums algunos valores.
        for(int i = 0; i < 10; i++)
            nums[i] = i;

        // Usa foreach para mostrar y sumar los valores.
        foreach(int x in nums) { Repetición cíclica a través de
            Console.WriteLine("El valor es: " + x);     nums utilizando el loop foreach.
            sum += x;
        }
        Console.WriteLine("Sumatoria: " + sum);
    }
}
```

Los datos de salida generados por el programa son:

```
El valor es: 0
El valor es: 1
El valor es: 2
El valor es: 3
El valor es: 4
El valor es: 5
El valor es: 6
El valor es: 7
El valor es: 8
El valor es: 9
Sumatoria: 45
```

Como lo muestran estos datos, **foreach** realiza ciclos repetitivos a través del arreglo en secuencia desde el índice inferior hasta el superior.

El loop **foreach** también funciona en arreglos multidimensionales. Regresa los elementos de acuerdo con el orden de las filas, de la primera a la última.

```
// Utiliza foreach en un arreglo bidimensional.

using System;

class ForeachDemo2 {
    static void Main() {
        int sum = 0;
        int[,] nums = new int[3,5];
```

```
// Da a nums algunos valores.  
for(int i = 0; i < 3; i++)  
    for(int j=0; j < 5; j++)  
        nums[i,j] = (i+1) * (j+1);  
  
// Usa foreach para mostrar y sumar los valores.  
foreach(int x in nums) {  
    Console.WriteLine("El valor es: " + x);  
    sum += x;  
}  
Console.WriteLine("Sumatoria: " + sum);  
}  
}
```

Los datos de salida que genera el programa son:

```
El valor es: 1  
El valor es: 2  
El valor es: 3  
El valor es: 4  
El valor es: 5  
El valor es: 2  
El valor es: 4  
El valor es: 6  
El valor es: 8  
El valor es: 10  
El valor es: 3  
El valor es: 6  
El valor es: 9  
El valor es: 12  
El valor es: 15  
Sumatoria: 90
```

Como **foreach** sólo puede hacer ciclos repetitivos a través de un arreglo de principio a fin, posiblemente pienses que su utilidad es limitada, pero no es así. Una gran cantidad de algoritmos requieren exactamente este mecanismo. Por ejemplo, a continuación presentamos una manera diferente de escribir la clase **MinMax**, que aparece en páginas anteriores de este capítulo y obtiene el mínimo y el máximo de un conjunto de valores:

```
/* Encuentra los valores mínimo y máximo en un arreglo  
utilizando un loop foreach. */  
  
using System;  
  
class MinMax {  
    static void Main() {  
        int[] nums = { 99, -10, 100123, 18, -978,  
                     5623, 463, -9, 287, 49 };  
        int min, max;
```

```

min = max = nums[0];
foreach(int val in nums) {
    if(val < min) min = val;
    if(val > max) max = val;
}
Console.WriteLine("Mínimo y máximo: " + min + " " + max);
}
}

```

El loop **foreach** es una excelente opción en esta aplicación porque encontrar los valores mínimo y máximo requiere examinar cada elemento. Otras aplicaciones de **foreach** incluyen tareas como calcular un promedio, buscar en una lista y copiar un arreglo.

## Cadenas de caracteres

Desde el punto de vista de la programación diaria, uno de los tipos de datos más importantes de C# es **string**. El tipo **string** define y soporta cadenas de caracteres. En muchos otros lenguajes de programación, **string** es un arreglo de caracteres, pero no es éste el caso en C#. En este lenguaje, las cadenas de caracteres son objetos, por lo que **string** es un tipo de referencia.

De hecho, has venido utilizando la clase **string** desde el capítulo 1 sin darte cuenta. Cuando creaste una literal de cadena, estabas creando de hecho un objeto **string**. Por ejemplo, en la declaración

```
Console.WriteLine("En C#, las cadenas son objetos.");
```

la cadena de caracteres “En C#, las cadenas son objetos.”, es convertida automáticamente en un objeto **string** por C#. De esta manera, el uso de la clase **string** ha actuado “tras bambalinas” en los pasados ejemplos. En esta sección aprenderás a manejarlas explícitamente. Ten presente, no obstante, que la clase **string** es muy amplia y aquí la compactaremos. Es una clase que querrás explorar por tu cuenta con mayor profundidad.

## Construir un objeto **string**

La manera más sencilla de construir un objeto **string** es utilizar una literal de caracteres. Por ejemplo, aquí **str** es una variable de referencia **string** asignada a una literal de cadena de caracteres:

```
string str = "Los objetos string de C# son poderosos.;"
```

En este caso, **str** se inicializa con la secuencia de caracteres “ Los objetos string de C# son poderosos.”

También puedes crear un **string** a partir de un arreglo **char**. Por ejemplo:

```
char[] chrs = {'t', 'e', 's', 't'};
string str = new string(chrs);
```

Una vez que has creado un objeto **string**, puedes utilizarlo en prácticamente cualquier lugar que acepte una literal de caracteres. Por ejemplo, puedes utilizar un objeto **string** como argumento para **WriteLine()**, como se muestra en el siguiente ejemplo:

```
// Introducción a string.  
  
using System;  
  
class StringDemo {  
    static void Main() {  
  
        char[] charray = {'A', ' ', 's', 't', 'r', 'i', 'n', 'g', '.'};  
        string str1 = new string(charray); ← Construye objetos string a partir  
        string str2 = "Otro string.";  
  
        Console.WriteLine(str1); ← Utiliza objetos string como  
        Console.WriteLine(str2); ← argumento para WriteLine().  
    }  
}
```

Los datos de salida que genera este programa son:

Un string.  
Otro string.

## Operadores de objetos **string**

La clase **string** contiene varios métodos que operan sobre las cadenas de caracteres. He aquí unas cuantas:

static string Copy(string str)	Regresa una copia de <i>str</i> .
int CompareTo(string str)	Regresa menos que cero si la cadena invocada es menor que <i>str</i> , mayor que cero si la cadena invocada es mayor que <i>str</i> , y cero si las cadenas son iguales.
int IndexOf(string str)	Busca en la cadena invocada la subcadena especificada por <i>str</i> . Regresa el índice de la <b>primera</b> coincidencia, o -1, en caso de falla.
int LastIndexOf(string str)	Busca en la cadena invocada la subcadena especificada por <i>str</i> . Regresa el índice de la <b>última</b> coincidencia, o -1, en caso de falla.

El tipo **string** también incluye la propiedad **Length**, que contiene la longitud de la cadena.

Para obtener el valor de un carácter individual en una cadena, simplemente utilizas un índice. Por ejemplo:

```
string str = "test";  
Console.WriteLine(str[0]);
```

Esto muestra la letra “t”. Como en los arreglos, los índices de las cadenas comienzan en cero. No obstante, un punto de importancia es que no puedes asignar un nuevo valor a un carácter dentro de la cadena utilizando un índice. Los índices sólo se pueden utilizar para obtener caracteres.

Para probar las coincidencias entre dos cadenas, puedes utilizar el operador `==`. Por lo regular, cuando se aplica el operador `==` a referencias de objetos, determina si ambas hacen referencia al mismo objeto. Con los objetos tipo **string** es diferente. Cuando el operador `==` se aplica a dos referencias **string**, el contenido de ambas se compara para encontrar coincidencias. Lo misma aplica al operador `!=`. Cuando se comparan objetos **string**, se comparan las cadenas de caracteres que contienen. Para realizar otro tipo de comparaciones necesitarás utilizar el método **CompareTo()**.

A continuación presentamos un programa que muestra varias operaciones **string**:

```
// Algunas operaciones string.

using System;

class StrOps {
    static void Main() {
        string str1 =
            "Cuando se trata de programar .NET, C# es #1.";
        string str2 = string.Copy(str1);
        string str3 = "En C# los strings son poderosos.";
        int result, idx;

        Console.WriteLine("Longitud de str1: " +
            str1.Length);

        // Muestra str1, un carácter a la vez.
        for(int i=0; i < str1.Length; i++)
            Console.Write(str1[i]);
        Console.WriteLine();

        if(str1 == str2)
            Console.WriteLine("str1 == str2");
        else
            Console.WriteLine("str1 != str2");

        if(str1 == str3)
            Console.WriteLine("str1 == str3");
        else
            Console.WriteLine("str1 != str3");

        result = str1.CompareTo(str3);
        if(result == 0)
            Console.WriteLine("str1 y str3 son iguales");
        else if(result < 0)
            Console.WriteLine("str1 es menor que str3");
        else
            Console.WriteLine("str1 es mayor que str3");

        // Asigna una nueva cadena a str2.
        str2 = "Uno Dos Tres Uno";
```

```
    idx = str2.IndexOf("Uno");
    Console.WriteLine("Índice de la primera ocurrencia de Uno: " + idx);
    idx = str2.LastIndexOf("Uno");
    Console.WriteLine("Índice de la última ocurrencia de Uno: " + idx);

}
```

El programa genera los siguientes datos de salida:

```
Longitud de str1: 44
Cuando se trata de programar .NET, C# es #1.
str1 == str2
str1 != str3
str1 es mayor que str3
Índice de la primera ocurrencia de Uno: 0
Índice de la última ocurrencia de Uno: 14
```

Puedes *concatenar* (unir) dos strings utilizando el operador **+**. Por ejemplo, la siguiente declaración:

```
string str1 = "Uno";
string str2 = "Dos";
string str3 = "Tres";
string str4 = str1 + str2 + str3;
```

inicializa **str4** con la cadena “Uno, Dos, Tres” como valor.

## Arreglos de objetos string

Como cualquier otro tipo de dato, los string pueden ser ensamblados en arreglos. Por ejemplo:

```
// Muestra un arreglo string.

using System;

class ArregloString {
    static void Main() {
        string[] str = { "Éste", "es", "un", "test." }; ← Un arreglo de cadenas.

        Console.WriteLine("Arreglo original: ");
        for(int i=0; i < str.Length; i++)
            Console.Write(str[i] + " ");
        Console.WriteLine("\n");

        // Cambia una cadena.
        str[1] = "era";
        str[3] = "test, también!";
```

```

Console.WriteLine("Arreglo modificado: ");
for(int i=0; i < str.Length; i++)
    Console.Write(str[i] + " ");
}
}

```

He aquí los datos de salida generados por el programa:

```

Arreglo original:
Éste es un test.

Arreglo modificado:
Éste era un test, también!

```

## Los objetos string son inmutables

He aquí algo que tal vez te sorprenda: los contenidos de un objeto **string** son inmutables. Es decir, una vez que ha sido creada, la secuencia de caracteres contenida en ese objeto **string** no puede ser alterada. Esta restricción permite que las cadenas de caracteres sean implementadas con mayor eficiencia. Aunque tal vez esto te parezca una contrariedad seria, en realidad no lo es. Cuando necesites una cadena que sea una variante sobre otra ya hecha, simplemente crea una nueva cadena que contenga los cambios que deseas. Dado que los objetos **string** que no se utilizan son eliminados por el recolector de basura, ni siquiera necesitas preocuparte por lo que sucede con las cadenas descartadas.

No obstante, debe quedar claro que las variables de referencia **string** pueden, por supuesto, cambiar el objeto al que se refieren. Lo que no puede modificarse una vez creado es el contenido de un objeto **string** específico.

Para comprender perfectamente por qué la inmutabilidad de objetos **string** no es un obstáculo, utilizaremos otro de sus métodos: **Substring()**. El método **Substring()** regresa una nueva cadena que contiene una porción específica del objeto **string** invocado. Dado que se crea un nuevo objeto **string** que contiene la subcadena, el contenido original no se altera y no se viola la regla de inmutabilidad. El formato de **Substring()** que utilizaremos se muestra a continuación:

```
string Substring(int índiceInicial, int largo)
```

Aquí, *índiceInicial* especifica el índice del primer carácter que será recuperado, y *largo* especifica la longitud de la subcadena.

Aquí tenemos un programa que muestra el uso de **Substring()** y el principio de inmutabilidad de los objetos **string**:

```

// Utiliza Substring().

using System;

class SubStr {
    static void Main() {
        string orgstr = "C# facilita el uso de strings.";

        // Construye una subcadena.
        string substr = orgstr.Substring(5, 12); ← Esto crea una nueva cadena que
                                                contiene la subcadena deseada.
    }
}

```

## Pregunta al experto

- P:** Usted afirma que una vez creados, los objetos `string` son inmutables. Comprendo que, desde un punto de vista práctico, no se trata de una restricción seria, ¿pero qué sucede si quiero crear una cadena de caracteres que sí sea modificable?
- R:** Tienes suerte. C# ofrece una clase llamada `StringBuilder` que se encuentra en la nomenclatura `System.Text`. Sirve para crear objetos `string` que pueden ser modificados. Sin embargo, para propósitos generales querrás utilizar `string` en lugar de `StringBuilder`.

```

    Console.WriteLine("orgstr: " + orgstr);
    Console.WriteLine("substr: " + substr);
}
}

```

He aquí los datos generados por el programa:

```

orgstr: C# facilita el uso de strings.
substr: facilita strings

```

Como puedes ver, la cadena original `orgstr` queda sin cambios y `substr` contiene la subcadena.

## Los operadores bitwise

En el capítulo 2 conociste los operadores aritméticos, relacionales y lógicos de C#. Ésos son los más utilizados, pero C# proporciona operadores adicionales que expanden el conjunto de problemas a los cuales puede ser aplicable este lenguaje: los operadores bitwise. Estos operadores actúan directamente sobre los bits de sus operandos. Están definidos para actuar sólo sobre operandos enteros. No pueden ser utilizados sobre tipos `bool`, `float`, `double` ni `class`.

Reciben el nombre de operadores *bitwise* porque se utilizan para probar, establecer o alterar los bits de los que consta un valor entero. Las operaciones bitwise son importantes en una gran cantidad de tareas de programación a nivel sistema: cuando, por ejemplo, debe obtenerse la información del estatus de un dispositivo o construir la misma. La tabla 5-1 muestra una lista de los operadores bitwise.

Operador	Resultado
&	Bitwise AND (Y)
	Bitwise OR (O)
^	Bitwise OR exclusivo (XOR)
>>	Traslado a la derecha
<<	Traslado a la izquierda
~	Complemento de uno (NOT unitario o Negación de)

**Tabla 5-1** Los operadores bitwise

## Los operadores bitwise AND, OR, XOR y NOT

Los operadores bitwise AND, OR, XOR y NOT son, respectivamente, **&**, **|**, **^** y **~**. Realizan las mismas operaciones que sus equivalentes booleanos descritos en el capítulo 2. La diferencia es que los operadores bitwise trabajan bit por bit. La siguiente tabla muestra el resultado de cada operación utilizando 1 y 0.

<b>p</b>	<b>q</b>	<b>p &amp; q</b>	<b>p   q</b>	<b>p ^ q</b>	<b>~p</b>
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

En términos de su uso más común, puedes pensar en el bitwise AND como una manera de apagar bits. Es decir, cualquier bit que es 0 en cualquier operando causará que el correspondiente bit en los datos de respuesta sea establecido en 0. Por ejemplo:

```

1 1 0 1 0 0 1 1
& 1 0 1 0 1 0 1 0
-----
1 0 0 0 0 0 1 0

```

El siguiente programa muestra el funcionamiento del operador **&** para convertir cualquier letra minúscula en mayúscula al restablecer el sexto bit a 0. En la manera como está establecido el conjunto de caracteres ASCII (que es un subconjunto de Unicode), las letras minúsculas son lo mismo que las mayúsculas, salvo que las primeras tienen un valor superior, exactamente 32. Por lo mismo, para transformar una minúscula en mayúscula sólo necesitas apagar el sexto bit, como lo ilustra el programa:

```

// Letras mayúsculas.

using System;

class Mayúsculas {
    static void Main() {
        char ch;

        for(int i=0; i < 10; i++) {
            ch = (char) ('a' + i);
            Console.Write(ch);

            // Esta declaración apaga el 6o. bit.
            ch = (char)(ch & 65503); // Ahora ch es mayúscula
            ↑
            Console.Write(ch + " ");
        }
    }
}


```

Utiliza el operador AND.

Los datos de salida que genera el programa son los siguientes:

```
aA bB cC dD eE fF gG hH iI jJ
```

El valor 65,503 utilizado en la operación AND es la representación decimal de 1111 1111 1101 1111. De manera que la operación AND deja intactos todos los bits de **ch**, con excepción del sexto, que se establece en cero.

El operador AND también es útil cuando quieras determinar si un bit está encendido o apagado. Por ejemplo, la siguiente declaración determina si el cuarto bit en **status** está encendido:

```
if((status & 8) != 0) Console.WriteLine("el bit 4 está encendido.");
```

La razón de utilizar el 8 es que traducido a valor binario sólo tiene el cuarto bit encendido. Por lo mismo, la declaración **if** sólo puede ser cierta cuando el cuarto bit de **status** está encendido. Un uso interesante de este concepto es mostrar los bits de un valor **byte** en formato binario:

```
// Muestra los bits dentro de un byte.  
  
using System;  
  
class BitsEnByte {  
    static void Main() {  
        byte val;  
  
        val = 123;  
        for(int t=128; t > 0; t = t/2) {  
            if((val & t) != 0) Console.Write("1 ");  
            else Console.Write("0 ");  
        }  
    }  
}
```

← Muestra los bits dentro del byte.

Los datos de salida son:

```
0 1 1 1 1 0 1 1
```

El loop **for** prueba exitosamente cada bit en **val**, utilizando el bitwise AND, para determinar si está encendido o apagado. Si el bit está encendido, se muestra el dígito **1**; en caso contrario se muestra **0**.

El bitwise OR puede utilizarse para encender bits. Cualquier bit establecido como 1 en cualquier operando causará que el correspondiente bit en la variable sea establecido como 1. Por ejemplo:

```
1 1 0 1 0 0 1 1  
| 1 0 1 0 1 0 1 0  
-----  
1 1 1 1 1 0 1 1
```

Podemos utilizar OR para convertir el programa de mayúsculas en minúsculas, como se muestra a continuación:

```
// Letras minúsculas.

using System;

class Minúsculas {
    static void Main() {
        char ch;

        for(int i=0; i < 10; i++) {
            ch = (char) ('A' + i);
            Console.WriteLine(ch);

            // Esta declaración enciende el 6o. bit.
            ch = (char) (ch | 32); // ahora ch es minúscula
            ↑
            Console.WriteLine(ch + " ");
        }
    }
}
```

Usa el operador OR.

Los datos de salida que genera el programa son:

Aa Bb Cc Dd Ee Ff Gg Hh Ii Jj

El programa funciona aplicando OR a cada carácter con el valor 32, que es 0000 0000 0010 0000 en binario. De esta manera, 32 es el valor decimal que produce un valor binario donde sólo el sexto bit está encendido. Cuando este valor es aplicado utilizando OR a cualquier otro valor, produce un resultado en el cual el sexto bit es encendido y el resto permanece intacto. Como se explicó anteriormente, para los caracteres ASCII esto significa que cada letra mayúscula se transforma en su equivalente en minúsculas.

El OR exclusivo, abreviado comúnmente XOR, encenderá un bit si, y sólo si, los bits que se comparan son diferentes, como se muestra a continuación:

0	1	1	1	1	1	1	1
^	1	0	1	1	0	0	1
<hr/>							
1	1	0	0	0	1	1	0

La operación XOR tiene una interesante propiedad que es útil en muy diversas situaciones. Cuando algún valor X es ajustado utilizando XOR con un valor Y, y al resultado se le vuelve a aplicar XOR de nuevo con Y, se produce X. Veamos un ejemplo. Dada la secuencia

R1 = X ^ Y;

R2 = R1 ^ Y;

R2 tiene el mismo valor que X. Así, el resultado de una secuencia de dos XOR utilizando el mismo valor comparativo, produce el valor original.

Para ver esta característica de XOR en acción, crearemos un programa de cifrado en el cual algún entero es la clave para cifrar y descifrar el mensaje aplicando XOR a todos los caracteres del mismo. Para la codificación, la operación XOR se aplica por primera vez, dando como resultado el texto cifrado. En la decodificación, XOR se aplica por segunda vez sobre el texto codificado, lo que resulta en el texto original. Por supuesto, este cifrado no tiene ningún valor práctico, porque es extremadamente fácil de romper. No obstante, representa una manera interesante de mostrar el funcionamiento de XOR, como lo muestra el siguiente programa:

```
// Muestra el funcionamiento de XOR.

using System;

class Codificador {
    static void Main() {
        string msg = "Ésta es una prueba";
        string encmsg = "";
        string decmsg = "";
        int key = 88;

        Console.WriteLine("Mensaje original: ");
        Console.WriteLine(msg);

        // Codifica el mensaje.
        for(int i=0; i < msg.Length; i++)
            encmsg = encmsg + (char) (msg[i] ^ key); ← Esto construye la cadena codificada.

        Console.WriteLine("Mensaje codificado: ");
        Console.WriteLine(encmsg);

        // Decodifica el mensaje.
        for(int i=0; i < msg.Length; i++)
            decmsg = decmsg + (char) (encmsg[i] ^ key); ← Esto construye la cadena decodificada.

        Console.WriteLine("Mensaje decodificado: ");
        Console.WriteLine(decmsg);
    }
}
```

Los datos de salida son:

```
Mensaje original: Ésta es una prueba
Mensaje codificado: 01+x1+x9x,=+
Mensaje decodificado: Ésta es una prueba
```

Como puedes ver, el resultado de dos XOR utilizando la misma clave produce la decodificación del mensaje. (Recuerda que este sencillo codificador XOR no es aplicable al mundo real, ni es de uso práctico, ya que es muy inseguro.)

El operador complemento unitario de 1 (NOT) invierte el estado de todos los bits del operando. Por ejemplo, si un entero llamado A tiene el patrón de bits 1001 0110, entonces ~A produce un resultado con el patrón de bits 0110 1001.

El siguiente programa muestra el operador NOT en acción al desplegar un número y su complemento en binario:

```
// Muestra el uso del bitwise NOT.

using System;

class NotDemo {
    static void Main() {
        sbyte b = -34;

        for(int t=128; t > 0; t = t/2) {
            if((b & t) != 0) Console.WriteLine("1 ");
            else Console.WriteLine("0 ");
        }
        Console.WriteLine();

        // Invierte todos los bits.
        b = (sbyte)~b;
        ↑                                     Utiliza el operador NOT.

        for(int t=128; t > 0; t = t/2) {
            if((b & t) != 0) Console.WriteLine("1 ");
            else Console.WriteLine("0 ");
        }
    }
}
```

Los datos de salida son:

```
1 1 0 1 1 1 1 0
0 0 1 0 0 0 0 1
```

## Los operadores de traslado

En C# es posible trasladar los bits que componen cierto valor, una cantidad específica de posiciones, hacia la derecha o hacia la izquierda. C# define los dos operadores de traslado que se muestran a continuación:

<<	Traslado a la izquierda
>>	Traslado a la derecha

Los formatos generales de estos operadores son:

*valor* << *num-bits*

*valor* >> *num-bits*

Aquí, *valor* es el valor que será trasladado la cantidad de posiciones de bit especificada por *num-bits*.

Un cambio hacia la izquierda causa que todos los bits dentro del valor especificado sean trasladados una posición hacia la izquierda y que sea traído un bit cero a la derecha. Un traslado hacia la derecha provoca que todos los bits sean movidos una posición hacia la derecha. En el caso del traslado hacia la derecha de un valor positivo, se coloca un cero a la izquierda. En el caso de un traslado a la derecha de un valor negativo, el signo del bit se conserva. Recuerda que todos los números negativos se representan estableciendo el bit de alto orden del entero en 1. De esta manera, si el valor que va a ser trasladado es negativo, cada movimiento hacia la derecha coloca un 1 a la izquierda. Si el valor es positivo, cada traslado a la derecha trae un cero a la izquierda.

En ambos trasladados, izquierda y derecha, los bits que se mueven se pierden. Así, los trasladados no son rotativos y no hay manera de recuperar un bit que ha sido trasladado fuera de la secuencia.

A continuación presentamos un programa que ilustra gráficamente el efecto del traslado a izquierda y derecha. A un entero se le da el valor inicial de 1, lo cual significa que se ha establecido su bit de bajo orden. Luego, se realiza una serie de ocho desplazamientos sobre el mismo entero. Despues de cada traslado los ocho bits bajos del valor son mostrados. A continuación, el proceso se repite, excepto que se coloca un 1 en la octava posición y se realizan los trasladados hacia la derecha.

```
// Muestra los operadores de traslado << y >>.

using System;

class TrasladoDemo {
    static void Main() {
        int val = 1;

        for(int i = 0; i < 8; i++) {
            for(int t=128; t > 0; t = t/2) {
                if((val & t) != 0) Console.Write("1 ");
                else Console.Write("0 ");
            }
            Console.WriteLine();
            val = val << 1; // traslado a la izquierda ← Traslado a la izquierda en val.
        }
        Console.WriteLine();

        val = 128;
        for(int i = 0; i < 8; i++) {
            for(int t=128; t > 0; t = t/2) {
                if((val & t) != 0) Console.Write("1 ");
                else Console.Write("0 ");
            }
            Console.WriteLine();
            val = val >> 1; // traslado a la derecha ← Traslado a la derecha en val.
        }
    }
}
```

## Pregunta al experto

**P:** Dado que los números binarios están basados en potencias de 2, ¿los operadores de traslado pueden ocuparse como atajo para multiplicar o dividir un entero entre 2?

**R:** Sí. Los operadores bitwise de traslado pueden utilizarse para realizar multiplicaciones y divisiones entre 2. Un traslado hacia la izquierda duplica un valor. Un traslado hacia la derecha lo divide a la mitad. Por supuesto, sólo funciona mientras no traslades bits fuera de la secuencia, de un lado o de otro.

Los datos de salida que genera el programa son los siguientes:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```

## Asignaciones para mezclar bitwise

Todas las operaciones binarias bitwise pueden utilizarse en asignaciones de mezcla. Por ejemplo, las siguientes dos declaraciones asignan a **x** el resultado de un XOR de **x** con valor de 127:

```
x = x ^ 127;
x ^= 127;
```

### Prueba esto

### Crear una clase que muestre los bits

Anteriormente en este mismo capítulo viste la clase **BitsEnByte** que muestra los bits que contiene un byte. Una limitante de **BitsEnByte** es que sólo funciona con valores **byte**. Una clase más útil mostraría los patrones de bits de cualquier tipo de valor entero. Por fortuna, una clase así es fácil de crear, y el programa que continúa muestra cómo hacerlo. Crea una clase llamada **Mues-**

(continúa)

**traBits** que permite mostrar los bits de cualquier tipo de entero positivo. También puedes utilizar **MuestraBits** para desplegar los bits en un valor entero negativo, pero necesitas transformarlo en su equivalente positivo. La función de ver los bits de cierto valor puede ser de gran utilidad en muchas situaciones. Por ejemplo, si recibes datos en bruto desde un dispositivo que son transmitidos a través de Internet, ver estos datos en su formato binario puede ayudarte a detectar un mal funcionamiento del dispositivo fuente.

## Paso a paso

1. Crea un archivo llamado **MuestraBitsDemo.cs**.
2. Comienza con la clase **MuestraBits**, como se muestra aquí:

```
class MuestraBits {
    public int numbits;

    public MuestraBits(int n) {
        numbits = n;
    }
}
```

**ShowBits** crea objetos que despliegan un número especificado de bits. Por ejemplo, para crear un objeto que desplegará ocho bits de control bajo de algún valor, usa

```
ShowBits b = new ShowBits(8);
```

El número de bits para desplegar se almacena en **numbits**.

3. Para desplegar de hecho el patrón de bits, **MuestraBits** proporciona el método **Muestra()**, que aparece a continuación:

```
public void Muestra(ulong val) {
    ulong mask = 1;

    // Desplaza a la izquierda un 1 a la posición adecuada.
    mask <<= numbits-1;

    int espaciador = 0;
    for(; mask != 0; mask >>= 1) {
        if((val & mask) != 0) Console.Write("1");
        else Console.Write("0");
        espaciador++;
        if((espaciador % 8) == 0) {
            Console.Write(" ");
            espaciador = 0;
        }
    }
    Console.WriteLine();
}
```

Observa que **Muestra()** especifica un parámetro **ulong**. Sin embargo, esto no significa que tengas que transmitir a **Muestra()** un valor **ulong**. Gracias a las conversiones implícitas de C#,

cualquier tipo de entero positivo puede ser transmitido a **Muestra()**. Para desplegar un valor entero negativo, tan sólo transfórmalo a su correspondiente tipo positivo. La cantidad de bits que se muestran está determinada por el valor almacenado en **numbits**. Después de cada grupo de ocho bits, **Muestra()** coloca un espacio en blanco. Esto facilita la lectura de valores binarios en patrones largos de bits.

4. El siguiente programa llamado **MuestraBitsDemo** pone en funcionamiento la clase **Muestra-Bits**:

```
// Una clase que muestra la representación binaria de un valor.

using System;

class MuestraBits {
    public int numbits;

    public MuestraBits(int n) {
        numbits = n;
    }

    public void Muestra(ulong val) {
        ulong mask = 1;

        // Desplaza a la izquierda un 1 a la posición adecuada.
        mask <=> numbits-1;

        int espaciador = 0;
        for(; mask != 0; mask >>= 1) {
            if((val & mask) != 0) Console.Write("1");
            else Console.Write("0");
            espaciador++;
            if((espaciador % 8) == 0) {
                Console.Write(" ");
                espaciador = 0;
            }
        }
        Console.WriteLine();
    }
}

// Funcionamiento de MuestraBits.
class MuestraBitsDemo {
    static void Main() {
        MuestraBits b = new MuestraBits(8);
        MuestraBits i = new MuestraBits(32);
        MuestraBits li = new MuestraBits(64);

        Console.WriteLine("123 en binario: ");
        b.Muestra(123);
```

(continúa)

```
Console.WriteLine("\n87987 en binario: ");
i.Muestra(87987);

Console.WriteLine("\n237658768 en binario: ");
li.Muestra(237658768);

// También puedes mostrar los bits de control bajo de cualquier
// entero.
Console.WriteLine("\n8 bits de Orden Bajo de 87987 en binario: ");
b.Muestra(87987);
}
```

5. Los datos de salida que genera **MuestraBitsDemo** son los siguientes:

123 en binario:  
01111011

87987 en binario:  
00000000 00000001 01010111 10110011

237658768 en binario:  
00000000 00000000 00000000 00000000 00001110 00101010 01100010  
10010000

8 bits de orden bajo de 87987 en binario:  
10110011

---

## El operador ?

Uno de los operadores más fascinantes de C# es ?, el operador condicional de este lenguaje. El operador ? es utilizado por lo regular para reemplazar la declaración **if-else** en el siguiente formato general:

```
if(condición)
    variable = expresión1;
else
    variable = expresión2;
```

Aquí, el valor asignado a *variable* depende del resultado de la *condición* que controla al **if**.

El operador ? recibe el nombre de *operador ternario* porque requiere tres operandos. Tiene el formato general siguiente:

*Exp1* ? *Exp2* : *Exp3*;

donde *Exp1* es una expresión **bool**, *Exp2* y *Exp3* son expresiones regulares que deben tener el mismo tipo de dato o por lo menos compatible. Observa el uso y la colocación de los dos puntos (:).

El valor de una expresión ? es determinado de la siguiente manera: *Exp1* es evaluada. Si es verdadera, entonces se evalúa *Exp2* y se convierte en el valor de la expresión ? completa. Si *Exp1* es falsa, entonces se evalúa *Exp3* y su valor se convierte en el de la expresión. Analiza el siguiente ejemplo, que asigna a **absval** el valor absoluto de **val**:

```
absval = val < 0 ? -val : val; // obtiene el valor absoluto de val
```

Aquí, a **absval** se le asignará el valor de **val** si **val** es cero o mayor. Si **val** es negativo, entonces a **absval** se le asignará el negativo de ese valor (el cual produce un valor positivo). El mismo código utilizando la estructura **if-else** se vería así:

```
if(val < 0) absval = -val;  
else absval = val;
```

He aquí otro ejemplo del operador ?. Este programa divide dos números, pero no permitirá una división entre cero:

```
// Previene una división entre cero utilizando el operador ?.  
  
using System;  
  
class NoCeroDiv {  
    static void Main() {  
        int resultado;  
  
        for(int i = -5; i < 6; i++) {  
            resultado = i != 0 ? 100 / i : 0; ← Esto previene una división entre cero.  
            if(i != 0)  
                Console.WriteLine("100 / " + i + " es " + resultado);  
        }  
    }  
}
```

Los datos de salida que genera este programa son los siguientes:

```
100 / -5 es -20  
100 / -4 es -25  
100 / -3 es -33  
100 / -2 es -50  
100 / -1 es -100  
100 / 1 es 100  
100 / 2 es 50  
100 / 3 es 33  
100 / 4 es 25  
100 / 5 es 20
```

Pon especial atención a esta línea del programa:

```
resultado = i != 0 ? 100 / i : 0;
```

En ella, a **resultado** se le asigna el producto de la división de 100 entre **i**. Sin embargo, la división sólo se realiza si el valor de **i** es diferente a cero. Cuando **i** es igual a cero, se le asigna a **resultado** un marcador de posición con ese mismo valor.

En realidad no tienes que asignar a una variable el valor producido por **?**. Por ejemplo, podrías utilizar el valor como argumento en la invocación de un método. O bien, si todas las expresiones son de tipo **bool**, el operador **?** puede ser utilizado como la expresión condicional en un loop de declaraciones **if**. Por ejemplo, a continuación aparece una versión más abreviada del programa anterior. Produce los mismos datos de salida.

```
// Previene una división entre cero utilizando el operador ?.

using System;

class NoCeroDiv2 {
    static void Main() {

        for(int i = -5; i < 6; i++)
            if(i != 0 ? true : false) ← Aquí, el operador ? se mueve
                Console.WriteLine("100 / " + i +
                    " es " + 100 / i);
    }
}
```

Observa la declaración **if**. Si **i** es cero, entonces el resultado de **if** es falso, la división entre cero no se realiza, y no se muestra ningún resultado. En cualquier otro caso, la división se lleva a cabo.



### Autoexamen Capítulo 5

1. Muestra cómo declarar un arreglo unidimensional de 12 **doubles**.
2. Muestra cómo declarar un arreglo bidimensional de 4 por 5 con valores **int**.
3. Muestra cómo declarar un arreglo descuadrado bidimensional con valores **int**, donde la primera dimensión sea 5.
4. Muestra cómo inicializar un arreglo unidimensional tipo **int** con valores del 1 al 5.
5. Explica qué es **foreach** y di cuál es su formato general.
6. Escribe un programa que utilice un arreglo para encontrar el promedio de diez valores **double**. Utiliza la decena de valores que gustes.
7. Cambia el orden *bubble* de la primera sección *Prueba esto*, de manera que organice un arreglo de cadenas de caracteres. Prueba que funcione.
8. ¿Cuál es la diferencia entre los métodos **IndexOf()** y **LastIndexOf()** del objeto **string**?
9. Expande la clase de cifrado **Codificador**, modifícalo de tal manera que utilice una cadena de ocho caracteres como clave.

**10.** ¿Se pueden aplicar los operadores bitwise al tipo **double**?

**11.** Muestra cómo puede ser expresada la siguiente secuencia utilizando el operador ?:

```
if(x < 0) y = 10;  
else y = 20;
```

**12.** En el siguiente fragmento, ¿es **&** un operador bitwise o es lógico? ¿Por qué?

```
bool a, b;  
// ...  
if(a & b) ...
```



# Capítulo 6

Un análisis profundo  
de los métodos  
y las clases

## Habilidades y conceptos clave

- Control de acceso de los miembros
  - Transmitir objetos a un método
  - Regresar objetos desde un método
  - Utilizar los parámetros **ref** y **out**
  - Sobrecargar métodos
  - Sobrecargar constructores
  - Regresar valores desde **Main()**
  - Transmitir argumentos a **Main()**
  - Recursión
  - El modificador **static**
- 

Este capítulo es una continuación de nuestra exploración sobre clases y métodos. Comienza explicando cómo controlar el acceso a los miembros de una clase. Después se enfoca en la transmisión y retorno de objetos, la sobrecarga de métodos, los diferentes formatos de **Main()**, la recursión y el uso de la palabra clave **static**.

## Controlar el acceso a los miembros de la clase

Con su soporte para la encapsulación, la clase proporciona dos grandes beneficios. Primero, vincula los datos con el código que los manipula. Has aprovechado este aspecto de la clase desde el capítulo 4. Segundo, proporciona los medios para controlar el acceso a los miembros de la clase. Este aspecto es el que se examina aquí.

Aunque el enfoque de C# es un poco más sofisticado, en esencia existen dos tipos básicos de miembros de clase: *public* (público) y *private* (privado). Un miembro *public* puede ser accesado libremente por el código definido fuera de su clase. Éste es el tipo de miembro de clase que hemos venido utilizando hasta este momento. Un miembro *private* puede ser accesado únicamente por otro método definido por su propia clase. El acceso se controla, precisamente, a través del uso de miembros privados.

Restringir el acceso a miembros de clase es parte fundamental de la programación orientada a objetos, porque ayuda a prevenir el mal uso de un objeto. Al permitir el acceso a datos privados sólo a través de un conjunto bien definido de métodos, puedes prevenir que se asignen valores equívocos a esos datos (al realizar una verificación de rango, por ejemplo). No es posible que el código definido fuera de la clase establezca directamente el valor de un miembro privado. También puedes controlar con precisión cómo y cuándo se usarán los datos dentro de un objeto. Así, cuando

se implementa de manera correcta, una clase crea una “caja negra”, que puede usarse, pero su contenido no está abierto para los fisiognomías.

## Especificadores de acceso de C#

El control de acceso a miembros de clase se consigue a través del uso de cuatro *especificadores de acceso*: **public** (público), **private** (privado), **protected** (protegido) e **internal** (interno). En este capítulo nos ocuparemos de **public** y **private**. El modificador **protected** se aplica sólo cuando está involucrada la herencia y se analiza en el capítulo 7. El modificador **internal** se aplica por lo común al uso de un *ensamblaje*, el cual en C# significa un programa, un proyecto o un componente. El modificador **internal** se explica brevemente en el capítulo 15.

Cuando un miembro de clase es modificado por el especificador **public**, puede ser accesado por cualquier otro código en el programa. Esto incluye métodos definidos en otras clases.

Cuando un miembro de clase se especifica como **private**, puede ser accesado sólo por otros miembros de su misma clase. De esta manera, los miembros de otras clases no pueden accesar un miembro **privado** de otra clase. Como se explicó en el capítulo 4, si no se utiliza ningún especificador, de forma predeterminada el miembro de clase se establece como privado de la clase a la que pertenece. Así, el especificador **private** es opcional cuando se crean miembros de clase privados.

Un especificador de acceso antecede al resto de la especificación de tipo del miembro de clase. Esto es, el especificador debe iniciar la declaración del miembro. He aquí algunos ejemplos:

```
public string ErrorMsg;
private double bal;
private bool isError(byte status) { // ...
```

Para comprender la diferencia entre **public** y **private**, analiza el siguiente programa:

```
// Acceso público contra privado.

using System;

class MiClase {
    private int alpha; // acceso privado especificado explícitamente
    int beta;         // acceso privado predeterminado
    public int gamma; // acceso público

/* Métodos para accesar alpha y beta. El miembro de una clase puede
   accesar un miembro privado de la misma clase. */

    public void SetAlpha(int a) {
        alpha = a;
    }

    public int GetAlpha() {
        return alpha;
    }

    public void SetBeta(int a) {
        beta = a;
    }
```

Estos miembros son privados  
de **MiClase**.

```
public int GetBeta() {
    return beta;
}

class AccesoDemo {
    static void Main() {
        MiClase ob = new MiClase();

        // El acceso a alpha y beta sólo es permitido a través de métodos.
        ob.SetAlpha(-99);
        ob.SetBeta(19);
        Console.WriteLine("ob.alpha es " + ob.GetAlpha());
        Console.WriteLine("ob.beta es " + ob.GetBeta());

        // No puedes accesar alpha o beta de la siguiente manera:
        // ob.alpha = 10; // ¡Error! ¡alpha es privado!
        // ob.beta = 9; // ¡Error! ¡beta es privado! ← Error! alpha y beta
                                                    son privados.

        // Es posible accesar gamma directamente porque es público.
        ob.gamma = 99; ← Correcto, porque gamma es público.
    }
}
```

Como puedes ver, dentro de la clase **MiClase**, **alpha** se especifica como **private** de manera explícita, **beta** es privado de forma predeterminada y **gamma** se especifica como **public**. Dado que **alpha** y **beta** son privados, no pueden ser accedidos fuera de su clase. Por lo mismo, dentro de la clase **AccesoDemo**, ninguno de ellos puede ser utilizado directamente. Cada uno debe ser accedido a través de métodos públicos, como **SetAlpha()** y **GetAlpha()**. Por ejemplo, si eliminaras el símbolo de comentario del inicio de la siguiente línea:

```
// ob.alpha = 10; // ¡Error! ¡alpha es privado!
```

no podrías compilar este programa debido a una violación de acceso. Aunque no está permitido el acceso a **alpha** por código fuera de **MiClase**, los métodos definidos dentro de **MiClase** tienen acceso libre, como lo demuestran los métodos **SetAlpha()** y **GetAlpha()**. Lo mismo se aplica a **beta**.

El punto clave es el siguiente: un miembro privado de una clase puede ser utilizado libremente por otros miembros de su clase, pero no puede ser accedido por código que esté fuera de su clase.

Para ver cómo puede ser utilizado el control de acceso en un ejemplo más práctico, analiza el siguiente programa que implementa un arreglo **int** de “falla leve”, donde los errores de desbordamiento son prevenidos, con lo que se evita una excepción en tiempo de error. Esto se consigue encapsulando el arreglo como miembro privado de una clase, con lo que el acceso queda restringido únicamente a los métodos de su misma clase. Con este enfoque se previene cualquier intento de accesar el arreglo más allá de sus límites, y tal intento falla con cierta gracia (el resultado es un aterrizaje “suave” en lugar de un “choque” brusco). El arreglo de falla leve es implementado por la clase **ArregloFallaLeve**, que se muestra a continuación:

```
/* Esta clase implementa un arreglo de "falla leve" que previene
   errores en tiempo de ejecución. */
```

```

using System;

class ArregloFallaLeve {
    private int[] a; // referencia al arreglo
    private int errval; // valor a regresar si Get() falla
    public int Longitud; // Longitud es público

    /* El constructor del arreglo determina su tamaño y el valor
       a regresar si Get() falla. */
    public ArregloFallaLeve(int tamaño, int errv) {
        a = new int[tamaño];
        errval = errv;
        Longitud = tamaño;
    }

    // Regresa valor a un índice dado.
    public int Get(int index) {
        if(ok(index)) return a[index];
        return errval;
    }

    // Coloca un valor en un índice. Regresa falso en caso de falla.
    public bool Put(int index, int val) {
        if(ok(index)) {
            a[index] = val;
            return true;
        }
        return false;
    }

    // Regresa verdadero si el índice está en los límites.
    private bool ok(int index) { ←
        if(index >= 0 & index < Longitud) return true;
        return false;
    }
}

// Muestra el arreglo de falla leve.
class FSDemo {
    static void Main() {
        ArregloFallaLeve fs = new ArregloFallaLeve(5, -1);
        int x;

        // Muestra las fallas silenciosas.
        Console.WriteLine("Falla silenciosa.");
        for(int i=0; i < (fs.Longitud * 2); i++)
            fs.Put(i, i*10);
        for(int i=0; i < (fs.Longitud * 2); i++) {
    
```

↑  
Observa que éstos son privados.

← Atrapa  
un índice  
fuera del  
límite.

← Un método privado.

```
x = fs.Get(i);
    if(x != -1) Console.WriteLine(x + " ");
}
Console.WriteLine("");

// Ahora, maneja las fallas.
Console.WriteLine("\nFalla con reportes de error.");
for(int i=0; i < (fs.Longitud * 2); i++)
    if(!fs.Put(i, i*10))
        Console.WriteLine("Índice " + i + " fuera de límite");

for(int i=0; i < (fs.Longitud * 2); i++) {
    x = fs.Get(i);
    if(x != -1) Console.WriteLine(x + " ");
    else
        Console.WriteLine("Índice " + i + " fuera de límite");
}
}
```

Los datos de salida generados por este programa son:

```
Falla silenciosa.
0 10 20 30 40

Falla con reportes de error.
Índice 5 fuera de límite
Índice 6 fuera de límite
Índice 7 fuera de límite
Índice 8 fuera de límite
Índice 9 fuera de límite
0 10 20 30 40 Índice 5 fuera de límite
Índice 6 fuera de límite
Índice 7 fuera de límite
Índice 8 fuera de límite
Índice 9 fuera de límite
```

Veamos de cerca este ejemplo. Dentro de **ArregloFallaLeve** están definidos tres miembros privados. El primero es **a**, que almacena una referencia al arreglo que contendrá, de hecho, la información. El segundo es **errval**, que es el valor que será regresado cuando falle la invocación a **Get()**. El tercer método privado es **ok()**, que determina si un índice está dentro de los límites. Así, estos tres miembros sólo pueden ser utilizados por otros miembros de **ArregloFallaLeve**. En forma específica, **a** y **errval** pueden ser usados sólo por otros métodos en la clase, y **ok()** puede ser usado sólo por otros miembros de **ArregloFallaLeve**. El resto de los miembros de clase son **public** y pueden ser invocados por cualquier otro código en el programa que utilice la clase **ArregloFallaLeve**.

Cuando se construye un objeto **ArregloFallaLeve**, debes especificar el tamaño del arreglo y el valor que quieras regresar en caso de que falle **Get()**. El valor de error debe ser uno que de otra manera no sea almacenado en el arreglo. Una vez construido, el arreglo en sí es referido como **a** y el valor del error almacenado en **errval** no puede ser accesado por los usuarios del objeto.

## Pregunta al experto

**P:** Si bien es cierto que la “falla leve” evita que el arreglo se desborde, lo hace a expensas de la sintaxis normal para la indexación de un arreglo. ¿Existe una manera mejor de crear un arreglo de “falla leve”?

**R:** Sí. Como verás en el capítulo 7, C# incluye un tipo especial de miembro de clase llamado *indexador*, que te permite indexar un objeto de clase como un arreglo. También hay una manera mejor de manejar el campo **Longitud** convirtiéndolo en una *propiedad*. Esto también se describe en el capítulo 7.

**ArregloFallaLeve.** De esta manera, queda cerrado a malos usos. Por ejemplo, el usuario no puede intentar indexar directamente **a**, lo que posiblemente excedería sus límites. El acceso sólo es permitido a través de los métodos **Get()** y **Put()**.

El método **ok()** es **privado**, más que nada, con fines ilustrativos. Sería inofensivo declararlo **public** porque no modifica el objeto. Sin embargo, como es utilizado internamente por la clase **ArregloFallaLeve**, puede ser declarado como **privado**.

Observa que la variable de instancia **Longitud** es **public**. Esto es consistente con la manera en que C# implementa el arreglo. Para obtener la longitud de **ArregloFallaLeve**, simplemente utiliza su miembro **Longitud**.

Para utilizar un arreglo **ArregloFallaLeve**, invoca **Put()** para almacenar un valor en el índice específico. Invoca **Get()** para recuperar un valor de un índice específico. Si el índice está fuera de límite, **Put()** regresa un valor **false** y **Get()** regresa **errval**.

Como los miembros de clase son privados de forma predeterminada, no hay razón para declararlos explícitamente utilizando la palabra **private**. Por lo mismo, desde este punto en adelante, en este libro no usaremos la declaración redundante **private** para establecer un miembro de clase como privado. Sólo recuerda que si un miembro de clase no está antecedido por un modificador de acceso, entonces es privado.

### Prueba esto

### Mejorar la clase sencilla de estructura en cola

Puedes utilizar el acceso privado para hacer una considerable mejora a la clase **ColaSimple** desarrollada en el capítulo 5. En aquella versión, todos los miembros de la clase **ColaSimple** eran públicos. Ello significa que un programa que utilizara esa clase podría accesar directamente el arreglo subyacente, y posiblemente tenga acceso a los elementos sin respetar el orden establecido. Dado que el objetivo esencial de una cola es proporcionar una lista ordenada como el primero que entra es el primero que sale, permitir el acceso en un orden diferente no es deseable. También es posible que un programador malicioso altere los valores almacenados en los índices **putloc** y **getloc**, corrompiendo así la cola. Por fortuna, este tipo de problemas se previenen con facilidad, haciendo privadas algunas partes de **ColaSimple**.

(continúa)

## Paso a paso

1. Copia el código original de **ColaSimple** a un nuevo archivo llamado **ColaSimple.cs**.
2. En la clase **ColaSimple**, elimina los especificadores **public** del arreglo **q** y de los índices **putloc** y **getloc**, como se muestra a continuación:

```
// Una clase cola mejorada para caracteres.
class ColaSimple {

    // Ahora estos elementos son privados.
    char[] q; // este arreglo almacena la cola
    int putloc, getloc; // los índices put y get

    public ColaSimple(int tamaño) {
        q = new char[tamaño+1]; // reserva memoria para la cola
        putloc = getloc = 0;
    }

    // Coloca un carácter en la cola.
    public void Put(char ch) {
        if(putloc==q.Length-1) {
            Console.WriteLine(" -- La cola está llena.");
            return;
        }

        putloc++;
        q[putloc] = ch;
    }

    // Obtiene un carácter de la cola.
    public char Get() {
        if(getloc == putloc) {
            Console.WriteLine(" -- La cola está vacía.");
            return (char) 0;
        }

        getloc++;
        return q[getloc];
    }
}
```

3. Modificar el acceso de **q**, **putloc** y **getloc** de público a privado no tiene efecto en un programa que utilice correctamente **ColaSimple**. Por ejemplo, todavía funciona correctamente con la clase **QDemo** del capítulo 5. Sin embargo, ahora previene el uso inapropiado de **ColaSimple**. Por ejemplo, las últimas dos declaraciones son ilegales:

```
ColaSimple test = new ColaSimple(10);
```

```
test.q[0] = 'X'; // ¡Error!
test.putloc = -100; // ¡No funcionará!
```

- 4.** Ahora que de **q**, **putloc** y **getloc** son privados, la clase **ColaSimple** se apega estrictamente al orden el primero que entra es el primero que sale.

## Transmitir una referencia de objeto a un método

Hasta este punto, los ejemplos presentados en este libro han utilizado tipos de valor, como **int** y **double**, como parámetros para los métodos. Sin embargo, es correcto y muy común transmitir la referencia de un objeto a un método. Por ejemplo, analiza el siguiente programa sencillo que almacena las dimensiones de un bloque tridimensional:

```
// Las referencias a objetos pueden ser transmitidas a métodos.

using System;

class Bloque {
    int a, b, c;
    int volumen;

    public Bloque(int i, int j, int k) {
        a = i;
        b = j;
        c = k;
        volumen = a * b * c;
    }

    // Regresa verdadero si ob define el mismo bloque.
    public bool MismoBloque(Bloque ob) {
        if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
        else return false;
    }

    // Regresa verdadero si ob tiene el mismo volumen.
    public bool MismoVolumen(Bloque ob) {
        if(ob.volumen == volumen) return true;
        else return false;
    }
}

class PasaOb {
    static void Main() {
        Bloque ob1 = new Bloque(10, 2, 5);
        Bloque ob2 = new Bloque(10, 2, 5);
        Bloque ob3 = new Bloque(4, 5, 5);
    }
}
```

Utiliza el tipo clase como parámetro.

```
Console.WriteLine("ob1 mismas dimensiones que ob2: " +  
                  ob1.MismoBloque(ob2)); ← Pasa una  
Console.WriteLine("ob1 mismas dimensiones que ob3: " +  
                  ob1.MismoBloque(ob3)); ← referencia  
Console.WriteLine("ob1 mismo volumen que ob3: " +  
                  ob1.MismoVolumen(ob3)); ← de objeto.  
}  
}
```

El programa genera los siguientes datos de salida:

```
ob1 mismas dimensiones que ob2: True  
ob1 mismas dimensiones que ob3: False  
ob1 mismo volumen que ob3: True
```

Los métodos **MismoBloque( )** y **MismoVolumen( )** comparan el objeto invocado con que se les transmitió como argumento. Para **MismoBloque( )**, las dimensiones del objeto son comparadas y regresa **true** sólo si los dos bloques son idénticos. Para **MismoVolumen( )**, los dos bloques son comparados sólo para determinar si tienen el mismo volumen. Observa que en ambos casos el parámetro **ob** especifica **Bloque** como un tipo. Como muestra este ejemplo, desde la perspectiva sintáctica, los tipos de referencia se transmiten a los métodos de la misma manera que los tipos de valor.

# Cómo se transmiten los argumentos

Como lo muestra el ejemplo anterior, transferir una referencia de objeto a un método es una tarea muy directa. Sin embargo, hay algunos matices que el ejemplo no muestra. En ciertos casos, el efecto de transmitir la referencia de un objeto será diferente de aquellos que se experimentan cuando se transmite un valor. Para ver por qué, necesitas entender los dos procedimientos a través de los cuales un argumento puede ser transmitido hacia una subrutina.

El primero lleva el nombre de *invocación-por-valor*. Este procedimiento copia el *valor* de un argumento al parámetro formal de la subrutina. Por lo mismo, los cambios hechos a los parámetros de la subrutina no tienen efecto sobre el argumento. El segundo procedimiento para transmitir un argumento lleva el nombre de *invocación-por-referencia*. En él, la *referencia* a un argumento (no el valor del argumento) es transmitida al parámetro. Dentro de la subrutina, esta referencia se utiliza para accesar el auténtico argumento especificado en la invocación. Esto significa que los cambios efectuados en el parámetro sí afectarán al argumento utilizado para invocar la subrutina.

De forma predeterminada, C# utiliza el procedimiento invocación-por-valor, lo que significa que se hace una copia del argumento y ésta es entregada al parámetro receptor. De esta manera, cuando transmises un tipo valor, como **int** o **double**, lo que ocurre al parámetro que recibe el argumento no tiene efecto fuera del método. Por ejemplo, analiza el siguiente programa:

```
// Los tipos simples son transmitidos por valor.  
  
using System;  
  
class Test {  
    // Este método no provoca cambios a los argumentos utilizados en la invocación.  
    public void NoCambio(int i, int j) {
```

```

    i = i + j;
    j = -j; ← Esto no cambia el argumento fuera de NoCambio().
}
}

class InvocaPorValor {
    static void Main() {
        Test ob = new Test();

        int a = 15, b = 20;

        Console.WriteLine("a y b antes de la invocación: " +
                          a + " " + b);

        ob.NoCambio(a, b);

        Console.WriteLine("a y b después de la invocación: " +
                          a + " " + b);
    }
}

```

Los datos de salida generados por este programa son los siguientes:

```

a y b antes de la invocación: 15 20
a y b después de la invocación: 15 20

```

Como puedes ver, la operación que ocurre dentro de **NoCambio()** no tienen ningún efecto en los valores **a** y **b** utilizados en la invocación.

Cuando transmites una referencia de objeto a un método, la situación es un poco más compleja. Técnicamente, la referencia de objeto en sí es transmitida por valor. De esta manera, se hace una copia de la referencia y los cambios hechos al parámetro no afectarán el argumento. Por ejemplo, hacer que el parámetro haga referencia a un nuevo objeto, no cambiará el objeto al cual se refiere el argumento. Sin embargo (y éste es un gran “sin embargo”), los cambios hechos *sobre el objeto* al que hace referencia el parámetro *sí afectarán* al objeto al que hace referencia el argumento, porque son uno y el mismo. Veamos por qué.

Recuerda que cuando creas una variable de tipo clase, estás creando una referencia a un objeto, no el objeto en sí. El objeto se crea en memoria a través de la palabra clave **new**, y la referencia a éste es asignada a la variable de referencia. Cuando utilizas una referencia de variable como argumento para un método, el parámetro recibe una referencia al mismo objeto al que hace referencia el argumento. Así, tanto el argumento como el parámetro hacen referencia al mismo objeto. Esto significa que los objetos son transmitidos a los métodos utilizando lo que es llamado, acertadamente, invocación-por-referencia. Los cambios al objeto dentro del método *sí afectan* al objeto utilizado como un argumento. Por ejemplo, analiza el siguiente programa:

```

// Los objetos son transmitidos implícitamente por referencia.

using System;

class Test {
    public int a, b;

```

```
public Test(int i, int j) {
    a = i;
    b = j;
}

/* Transmite un objeto. Ahora, ob.a y ob.b en el objeto
   usado en la invocación serán modificados. */
public void Cambio(Test ob) {
    ob.a = ob.a + ob.b; ← Esto cambiará los argumentos.
    ob.b = -ob.b;
}

class InvocaPorRef {
    static void Main() {
        Test ob = new Test(15, 20);

        Console.WriteLine("ob.a y ob.b antes de la invocación: " +
                           ob.a + " " + ob.b);
        ob.Cambio(ob);

        Console.WriteLine("ob.a y ob.b después de la invocación: " +
                           ob.a + " " + ob.b);
    }
}
```

Este programa genera los siguientes datos de salida:

```
ob.a y ob.b antes de la invocación: 15 20
ob.a y ob.b después de la invocación: 35 -20
```

Como puedes ver, las acciones dentro de **Cambio()** han afectado al objeto utilizado como argumento.

Para resumir: cuando una referencia de objeto es transmitida a un método, la referencia en sí pasa utilizando la invocación-por-valor. Así que se hace una copia de dicha referencia. No obstante, como el valor transmitido hace referencia a un objeto, la copia de ese valor seguirá haciendo referencia al mismo objeto que su correspondiente argumento.

## Utilizar los parámetros **ref** y **out**

Como acabamos de explicar, los tipos de valor como **int** y **char**, predeterminadamente, son transmitidos por valor a un método. Esto significa que los cambios realizados al parámetro que recibe un tipo valor no afectarán el argumento real utilizado en la invocación. Sin embargo, puedes alterar este comportamiento. Con el uso de las palabras clave **ref** y **out**, es posible transmitir cualquiera de los tipos de valor por referencia. Al hacerlo, se permite que el método altere el argumento utilizado en la invocación.

Antes de estudiar los mecanismos que utilizan **ref** y **out**, es de utilidad comprender por qué querrías transmitir un tipo de valor como referencia. En general existen dos razones: permitir que el método altere el contenido de sus argumentos o permitir que un método regrese más de un valor. Veamos con detalle cada razón.

Por lo regular, querrás que un método sea capaz de operar sobre los argumentos que le son transmitidos. El ejemplo por excelencia de esto es un método “de intercambio” que, como su nombre lo indica, intercambia los valores de sus dos argumentos. Como los tipos de valor son transmitidos por valor, no es posible escribir un método tal que intercambie el valor de dos **int**, por ejemplo, utilizando el mecanismo de forma predeterminada de C# invocación-por-valor para transmitir parámetros. El modificador **ref** resuelve este problema.

Como sabes, una declaración **return** permite que un método regrese un valor a su invocador. Sin embargo, un método puede regresar *sólo un valor* cada vez que se invoca. ¿Qué sucede si necesitas regresar dos o más piezas de información? Por ejemplo, ¿qué tal si quieres crear un método que calcule el área de un cuadrilátero y que también determine si es un cuadrado? Tal tarea requiere que regresen dos piezas de información: el área y un valor que indica la cuadratura. Este método no puede ser escrito utilizando un solo valor de regreso. El modificador **out** resuelve este problema.

## Usar **ref**

El modificador de parámetro **ref** provoca que C# cree una invocación-por-referencia en lugar de una invocación-por-valor. El modificador **ref** se aplica cuando un método es declarado y cuando es invocado. Comencemos con un ejemplo sencillo. El siguiente programa crea un método llamado **Cuadrado()** que regresa el cuadrado de su argumento, que es un número entero. Observa el uso y emplazamiento de **ref**.

```
// Usa ref para transmitir un tipo de valor por referencia.

using System;

class RefTest {
    // Este método cambia sus argumentos.
    public void Cuadrado(ref int i) { ← Aquí, ref antecede a la declaración del parámetro.
        i = i * i;
    }
}

class RefDemo {
    static void Main() {
        RefTest ob = new RefTest();

        int a = 10;

        Console.WriteLine("a antes de la invocación: " + a);

        ob.Cuadrado(ref a); ← Aquí, ref antecede al argumento.

        Console.WriteLine("a después de la invocación: " + a);
    }
}
```

Observa que **ref** antecede la declaración de parámetro completa en el método y también antecede el nombre del argumento cuando el método es invocado. Los datos de salida que genera este programa, mostrados a continuación, confirman que el valor del argumento, **a**, es en efecto modificado por **Cuadrado()**:

```
a antes de la invocación: 10  
a después de la invocación: 100
```

Ahora es posible utilizar **ref** para escribir un método que intercambie los valores de sus dos argumentos tipo valor. Por ejemplo, a continuación presentamos un programa que contiene un método llamado **Intercambio()** que intercambia los valores de los dos argumentos que son utilizados en su invocación; ambos son números enteros:

```
// Intercambia dos valores.  
  
using System;  
  
class IntercambioDemo {  
    // Este método intercambia sus argumentos.  
    public void Intercambio(ref int a, ref int b) {  
        int t;  
  
        t = a;  
        a = b;  
        b = t;  
    }  
}  
  
class IntercambiaLos {  
    static void Main() {  
        IntercambioDemo ob = new IntercambioDemo();  
  
        int x = 10, y = 20;  
  
        Console.WriteLine("x y y antes de la invocación: " + x + " " + y);  
  
        ob.Intercambio(ref x, ref y);  
  
        Console.WriteLine("x y y después de la invocación: " + x + " " + y);  
    }  
}
```

Los datos de salida generados por el programa son:

```
x y y antes de la invocación: 10 20  
x y y después de la invocación: 20 10
```

Aquí hay un punto importante para comprender sobre **ref**: un argumento transferido por **ref** debe tener un valor asignado previamente a la invocación. La razón es que el método que recibe tal argumento da por hecho que el parámetro hace referencia a un valor legítimo. Así, al aplicar **ref** no puedes utilizar un método que le dé un valor inicial al argumento.

## Usar out

Algunas ocasiones querrás utilizar un parámetro de referencia que reciba el valor de un método, pero que no transmita un valor. Por ejemplo, puedes tener un método que realice algunas funciones, como abrir un enlace de red, que regrese un código de éxito/falla en un parámetro de referencia. En este caso, no hay información para transmitir al método, pero sí hay información para transmitir de vuelta. El problema en este escenario es que un parámetro **ref** debe ser inicializado con un valor antes de ser invocado. De esta manera, el parámetro **ref** requeriría dar un valor poste-  
rizo al argumento, sólo para satisfacer esta restricción. Por fortuna, C# proporciona una mejor alter-  
nativa: el parámetro **out**.

Un parámetro **out** es similar a uno **ref** con una excepción: sólo puede ser utilizado para trans-  
mitir un valor fuera de un método. No es necesario (o útil) dar un valor inicial a la variable utili-  
zada como **out** antes de invocar el método. El método mismo dará un valor a la variable. Más aún,  
dentro del método, un parámetro **out** siempre es considerado *sin asignación*; es decir, se da por  
hecho que no cuenta con un valor inicial. En lugar de ello, el método *debe* asignarle al parámetro  
un valor antes de concluir. Así, después de la invocación al método, la variable a la que hace refe-  
rencia a través del parámetro **out** contendrá un valor.

A continuación presentamos un ejemplo que utiliza el parámetro **out**. El método **RectInfo()**  
regresa el área de un rectángulo, dada la longitud de sus lados. En el parámetro **esCuadrado**, re-  
gresa **true** si el rectángulo tiene la misma longitud en todos sus lados y **false** en caso contrario. De  
esta manera, **RectInfo()** regresa dos piezas de información al invocador.

```
// Usa un parámetro out.

using System;

class Rectángulo {
    int lado1;
    int lado2;

    public Rectángulo(int i, int j) {
        lado1 = i;
        lado2 = j;
    }

    // Regresa el área y determina si es un cuadrado.
    public int RectInfo(out bool esCuadrado) { ← Transmite información fuera del
        if(lado1==lado2) esCuadrado = true; método a través del parámetro out.
        else esCuadrado = false;

        return lado1 * lado2;
    }
}

class OutDemo {
    static void Main() {
        Rectángulo rect = new Rectángulo(10, 23);
```

```
int área;
bool esCuadrado;

área = rect.RectInfo(out esCuadrado);

if(esCuadrado) Console.WriteLine("rect es un cuadrado.");
else Console.WriteLine("rect no es un cuadrado.");

Console.WriteLine("Su área es " + área + ".");
}
```

Observa que a **esCuadrado** no se le asigna ningún valor antes de la invocación a **RectInfo()**. Esto no sería permitido si el parámetro a **RectInfo()** hubiese sido **ref** en lugar de **out**. Después de que el método regresa, **esCuadrado** contiene un valor **true** o **false**, dependiendo de si los lados del rectángulo son o no iguales. El área es regresada a través de la declaración **return**. Los datos de salida generados por este programa son:

```
rect no es un cuadrado.
Su área es 230.
```

### Pregunta al experto

**P:** ¿Es posible utilizar **ref** y **out** en parámetros de tipo referencia, como cuando se transmite la referencia a un objeto?

**R:** Sí. Cuando **ref** y **out** modifican un parámetro tipo referencia, hacen que ésta, en sí, sea transmitida por referencia. Esto permite que lo referido sea modificado por el método. Analiza el siguiente programa:

```
// Utiliza ref sobre un parámetro objeto.
using System;

class Test {
    public int a;

    public Test(int i) {
        a = i;
    }
    // Esto no modificará el argumento.
    public void NoCambio(Test o) {
        Test newobj = new Test(0);
        o = newobj; // Esto no tiene efecto fuera de NoCambio()
    }
}
```

```
// Esto cambiará lo referido por el argumento.
public void Cambio(ref Test o) {
    Test newob = new Test(0);
    o = newob; // esto afecta el argumento invocador.
}

class InvocaObjPorRef {
    static void Main() {
        Test ob = new Test(100);

        Console.WriteLine("ob.a antes de la invocación: " + ob.a);

        ob.NoCambio(ob);
        Console.WriteLine("ob.a después de la invocación a NoCambio(): " +
ob.a);

        ob.Cambio(ref ob);
        Console.WriteLine("ob.a después de la invocación a Cambio(): " +
ob.a);
    }
}
```

Los datos de salida generados por el programa son:

```
ob.a antes de la invocación: 100
ob.a después de la invocación a NoCambio(): 100
ob.a después de la invocación a Cambio(): 0
```

Como puedes ver, cuando a **o** se le asigna una referencia a un nuevo objeto dentro de **NoCambio()**, no hay ningún efecto sobre el argumento **ob** dentro de **Main()**. Sin embargo, dentro de **Cambio()**, que utiliza un parámetro **ref**, asignar un nuevo objeto a **o** sí cambia el objeto referenciado por **ob** dentro de **Main()**.

## Utilizar un número indeterminado de argumentos

Cuando creas un método, por lo regular sabes por adelantado la cantidad de argumentos que le transmitirás, pero no siempre sucede así. Algunas veces querrás crear un método al que le pueden ser transmitidos una cantidad arbitraria de argumentos. Por ejemplo, considera un método que encuentre el valor más pequeño en un conjunto. Un método así podría recibir al menos dos valores, pero también podrían ser tres o cuatro y así sucesivamente. En todos los casos querrías que el método regresara el valor más pequeño. Un método con estas características no puede crearse utilizando parámetros normales. En vez de eso, debes utilizar un tipo especial de parámetro que represente una cantidad arbitraria de los mismos. Esto se consigue creando una entidad **params**.

El modificador **params** se utiliza para declarar un arreglo de parámetros que podrán recibir cero o más argumentos. La cantidad de elementos en el arreglo será igual al número de argumentos transmitidos al método. Entonces, tu programa accesa el arreglo para obtener los argumentos.

He aquí un ejemplo que utiliza **params** para crear un método llamado **MinVal()**, que regresa el valor mínimo de un conjunto:

```
// Muestra params.

using System;

class Min {
    public int MinVal(params int[] nums) { ← Crea un parámetro de longitud
        int m;

        if(nums.Length == 0) {
            Console.WriteLine("Error: no hay argumentos.");
            return 0;
        }

        m = nums[0];
        for(int i=1; i < nums.Length; i++)
            if(nums[i] < m) m = nums[i];

        return m;
    }
}

class ParamsDemo {
    static void Main() {
        Min ob = new Min();
        int min;
        int a = 10, b = 20;

        // Invocación con dos valores.
        min = ob.MinVal(a, b);
        Console.WriteLine("Mínimo es " + min);

        // Invocación con 3 valores.
        min = ob.MinVal(a, b, -1);
        Console.WriteLine("Mínimo es " + min);

        // Invocación con 5 valores.
        min = ob.MinVal(18, 23, 3, 14, 25);
        Console.WriteLine("Mínimo es " + min);

        // También se puede invocar con un arreglo de int.
        int[] args = { 45, 67, 34, 9, 112, 8 };
        min = ob.MinVal(args);
        Console.WriteLine("Mínimo es " + min);
    }
}
```

Los datos de salida generados por el programa:

```
Mínimo es 10
Mínimo es -1
Mínimo es 3
Mínimo es 8
```

Cada vez que se invoca **MinVal()**, los argumentos le son transmitidos a través del arreglo **nums**. La longitud del arreglo es igual al número de elementos. Así, puedes utilizar **MinVal()** para encontrar el mínimo en cualquier cantidad de valores.

Aunque puedes transmitir a **params** cualquier cantidad de argumentos, todos ellos deben ser de un tipo compatible con el del arreglo, especificado por el parámetro. Por ejemplo, invocar **MinVal()** de esta manera:

```
min = ob.MinValue(1, 2.2);
```

es ilegal porque no hay una conversión implícita de **double** (2.2) a **int**, que es el tipo de **nums** en **MinVal()**.

Cuando se utiliza **params**, debes tener cuidado sobre las condiciones de los límites porque el parámetro **params** puede aceptar cualquier cantidad de argumentos, *¡incluso cero!* Por ejemplo, sintácticamente es válido invocar **MinVal()** como se muestra a continuación:

```
min = ob.MinValue(); // sin argumentos
min = ob.MinValue(3); // 1 argumento
```

Por ello existe una verificación en **MinVal()** para confirmar que el arreglo **nums** tiene al menos un elemento, antes de intentar el acceso. Si la verificación no estuviera ahí, daría como resultado una excepción en tiempo de ejecución si **MinVal()** fuera invocado sin argumentos. (Más adelante en este mismo libro, cuando se aborden las excepciones, verás una mejor manera de manejar este tipo de errores.) Más aún, el código en **MinVal()** fue escrito de tal manera que permite su invocación con un argumento. En esa situación, se regresa ese único elemento.

Un método puede tener parámetros normales y un parámetro de longitud indeterminada. Por ejemplo, en el siguiente programa, el método **MuestraArgs()** toma un parámetro **string** y luego un arreglo de enteros **params**:

```
// Utiliza parámetros regulares con un parámetro params.

using System;
class MiClase {
    public void MuestraArgs(string msg, params int[] nums) {
        Console.WriteLine(msg + ": ");
        foreach (int i in nums)
            Console.Write(i + " ");
        Console.WriteLine();
    }
}
```

Este método tiene un parámetro normal y uno **params**.



```
        }

class ParamsDemo2 {
    static void Main() {
        MiClase ob = new MiClase();

        ob.MuestraArgs("He aquí algunos enteros",
                       1, 2, 3, 4, 5);
        ob.MuestraArgs("Aquí hay dos más",
                       17, 20);

    }
}
```

El programa genera los siguientes datos de salida:

```
He aquí algunos enteros: 1 2 3 4 5
Aquí hay dos más: 17 20
```

En casos en que el método tenga parámetros regulares y **params**, éste debe ser el último en la lista de parámetros. Más aún, en todas las situaciones sólo puede existir un solo parámetro **params**.

## Objetos como respuesta

Un método puede regresar cualquier tipo de dato, incluyendo el tipo clase. Por ejemplo, la clase **ErrorMsg** puede ser utilizada para reportar errores. Su método **GetErrorMsg( )** regresa un objeto **string** que contiene la descripción del error, con base en el código de error que le es transmitido.

```
// Regresa un objeto string.

using System;

class ErrorMsg {
    string[] msgs = {
        "Error de salida",
        "Error de entrada",
        "Disco Lleno",
        "Índice Fuera de Límites"
    };

    // Regresa el mensaje de error.
    public string GetErrorMsg(int i) { ←———— Regresa un objeto de tipo string.
        if(i >=0 & i < msgs.Length)
            return msgs[i];
        else
    }
}
```

```
        return "Código de Error Inválido";
    }
}

class ErrMsg {
    static void Main() {
        ErrMsg err = new ErrMsg();
        Console.WriteLine(err.GetErrorMsg(2));
        Console.WriteLine(err.GetErrorMsg(19));
    }
}
```

Los datos de salida generados por el programa son:

```
Disco Lleno
Código de Error Inválido
```

Por supuesto, también puedes regresar objetos de clases que tú mismo crees. Por ejemplo, a continuación presentamos una versión modificada del programa anterior que crea dos clases de error. Una es llamada **Err**, y encapsula un mensaje de error junto con un código de gravedad. El segundo recibe el nombre de **ErrorInfo** y define un método llamado **GetInfoError()** que regresa un objeto **Err**.

```
// Regresa un objeto definido por el programador.

using System;

class Err {
    public string Msg; // mensaje de error
    public int Severidad; // código que indica la gravedad del error

    public Err(string m, int s) {
        Msg = m;
        Severidad = s;
    }
}

class ErrorInfo {
    string[] msgs = {
        "Error de salida",
        "Error de entrada",
        "Disco Lleno",
        "Índice Fuera de Límites"
    };
    int[] grado = { 3, 3, 2, 4 };

    public Err GetErrorInfo(int i) {————— Regresa un objeto tipo Err.
        if(i >= 0 & i < msgs.Length)
```

```
        return new Err(msgs[i], grado[i]);
    else
        return new Err("Código de Error Inválido", 0);
}
}

class ErrInfo {
    static void Main() {
        ErrorInfo err = new ErrorInfo();
        Err e;

        e = err.GetErrorInfo(2);
        Console.WriteLine(e.Msg + " gravedad: " + e.Severidad);

        e = err.GetErrorInfo(19);
        Console.WriteLine(e.Msg + " gravedad: " + e.Severidad);
    }
}
```

Los datos de salida generados por el programa:

```
Disco Lleno gravedad: 2
Código de Error Inválido gravedad: 0
```

Cada vez que se invoca **GetErrorInfo()**, se crea un nuevo objeto **Err** y una referencia de él es regresada a la rutina invocadora. Después, este objeto es utilizado por **Main()** para mostrar el mensaje de error y el código de gravedad.

Cuando un objeto es regresado por un método, existe hasta que no hay más referencias sobre él. En ese momento pasa a ser material para el recolector de basura. Así, un objeto no será destruido sólo porque finalice el método que lo creó.

## Sobrecargar un método

En esta sección aprenderás una de las características más emocionantes de C#: la recarga de métodos. En C#, dos o más métodos dentro de la misma clase pueden compartir el mismo nombre, siempre y cuando las declaraciones de sus parámetros sean diferentes. Cuando sucede esto, se dice que los métodos están *sobrecargados* y el método para completar esta tarea recibe el nombre de *sobre carga de métodos*. Ésta es una de las maneras como C# implementa el polimorfismo.

En general, para sobrecargar un método simplemente declara diferentes versiones del mismo. El compilador se ocupa del resto. Debes observar una importante restricción: el tipo o el número de los parámetros de cada método sobrecargado debe ser diferente. No es suficiente que ambos métodos sólo se diferencien en sus tipos de regreso. Deben ser diferentes en el tipo o número de sus parámetros. (Los tipos de regreso no proporcionan suficiente información en todos los casos para que C# decida cuál método utilizar.) Por supuesto, los métodos sobrecargados también *pueden* ser diferentes en sus tipos de regreso. Cuando se invoca un método sobrecargado, se ejecuta la versión de éste cuyos parámetros coinciden con los argumentos.

He aquí un ejemplo que ilustra la sobrecarga de métodos:

```
// Muestra la sobrecarga de métodos.

using System;

class Sobrecarga {
    public void SCDemo() {————Primera versión.
        Console.WriteLine("Sin parámetros");
    }

    // Sobrecarga SCDemo para un parámetro de enteros.
    public void SCDemo(int a) {————Segunda versión.
        Console.WriteLine("Un parámetro: " + a);
    }

    // Sobrecarga SCDemo para dos parámetros de entero.
    public int SCDemo(int a, int b) {————Tercera versión.
        Console.WriteLine("Dos parámetros: " + a + " " + b);
        return a + b;
    }

    // Sobrecarga SCDemo para dos parámetros double.
    public double SCDemo(double a, double b) {————Cuarta versión.
        Console.WriteLine("Dos parámetros double: " +
            a + " " + b);
        return a + b;
    }
}

class SobreCargaDemo {
    static void Main() {
        Sobrecarga ob = new Sobrecarga();
        int resI;
        double resD;

        // Invoca todas las versiones de SCDemo().
        ob.SCDemo();
        Console.WriteLine();

        ob.SCDemo(2);
        Console.WriteLine();

        resI = ob.SCDemo(4, 6);
        Console.WriteLine("Resultado de ob.SCDemo(4, 6): " +
            resI);
        Console.WriteLine();

        resD = ob.SCDemo(1.1, 2.32);
```

```
        Console.WriteLine("Resultado de ob.SCDemo(1.1, 2.2): " +
                           resD);
    }
}
```

El programa genera los siguientes datos de salida:

Sin parámetros

Un parámetro: 2

Dos parámetros: 4 6

Resultado de ob.SCDemo(4, 6): 10

Dos parámetros double: 1.1 2.2

Resultado de ob.SCDemo(1.1, 2.2): 3.42

Como puedes ver **SCDemo()** se sobrecarga cuatro veces. En la primera versión no hay parámetros, la segunda toma un parámetro de número entero, la tercera dos parámetros de números enteros y la cuarta dos parámetros **double**. Observa que las primeras dos versiones de **SCDemo()** regresan **void** (vacío) y las dos restantes un valor. Esto es perfectamente válido, pero como se explicó, la sobrecarga no se ve afectada de ninguna manera por el tipo de retorno del método. Por ello, intentar usar las siguientes dos versiones de **SCDemo()** provocaría un error.

```
// Un SCDemo(int) es correcto.
public void SCDemo(int a) { ←
    Console.WriteLine("Un parámetro: " + a);
}

/* ¡Error! Dos SCDemo(int) no son correctos aunque
   los tipos de regreso sean diferentes. */
public int SCDemo(int a) { ←
    Console.WriteLine("Un parámetro: " + a);
    return a * a;
}
```

Los tipos de regreso no  
pueden ser utilizados  
para diferenciar métodos  
sobrecargados.

Como lo sugieren los comentarios, la diferencia entre tipos de regreso no es suficiente para diferenciar el propósito de la sobrecarga.

Como recordarás del capítulo 2, C# proporciona ciertas conversiones implícitas de tipo. Estas conversiones también se aplican a los parámetros de métodos sobrecargados. Por ejemplo, analiza el siguiente programa:

```
// Conversiones implícitas de tipo pueden afectar el resultado de un
// método recargado.

using System;

class Sobrecarga2 {
    public void MiMétodo(int x) {
        Console.WriteLine("Dentro de MiMétodo(int): " + x);
    }
}
```

```

public void MiMétodo(double x) {
    Console.WriteLine("Dentro de MiMétodo(double): " + x);
}
}

class ConvTipo {
    static void Main() {
        Sobrecarga2 ob = new Sobrecarga2();

        int i = 10;
        double d = 10.1;

        byte b = 99;
        short s = 10;
        float f = 11.5F;

        ob.MiMétodo(i); // invoca ob.MiMétodo(int)
        ob.MiMétodo(d); // invoca ob.MiMétodo(double)

        ob.MiMétodo(b); // invoca ob.MiMétodo(int) -- conversión de tipo
        ob.MiMétodo(s); // invoca ob.MiMétodo(int) -- conversión de tipo
        ob.MiMétodo(f); // invoca ob.MiMétodo(double) -- conversión de tipo
    }
}

```

Los datos de salida generados por el programa son:

```

Dentro de MiMétodo(int): 10
Dentro de MiMétodo(double): 10.1
Dentro de MiMétodo(int): 99
Dentro de MiMétodo(int): 10
Dentro de MiMétodo(double): 11.5

```

En este ejemplo, sólo dos versiones de **MiMétodo()** son definidas: una que tiene un parámetro **int** y otra que tiene un parámetro **double**. Sin embargo, es posible transmitir a **MiMétodo()** un valor **byte**, **short** o **float**. En el caso de **byte** y **short**, C# los convierte automáticamente en **int**. De esta manera, **MiMétodo(int)** es invocado. En el caso de **float**, el valor se convierte en **double** y se invoca **MiMétodo(double)**.

Es importante comprender, no obstante, que la conversión implícita aplica sólo si no existe una correspondencia directa entre los tipos de un parámetro y un argumento. Por ejemplo, a continuación presentamos una versión nueva del programa anterior en la que se añade una versión de **MiMétodo()** que especifica un parámetro **byte**:

```

// Añade MiMétodo(byte).

using System;

class Sobrecarga2 {
    public void MiMétodo(byte x) {

```

```
        Console.WriteLine("Dentro de MiMétodo(byte): " + x);
    }

    public void MiMétodo(int x) {
        Console.WriteLine("Dentro de MiMétodo(int): " + x);
    }

    public void MiMétodo(double x) {
        Console.WriteLine("Dentro de MiMétodo(double): " + x);
    }
}

class ConvTipo {
    static void Main() {
        Sobrecarga2 ob = new Sobrecarga2();

        int i = 10;
        double d = 10.1;

        byte b = 99;
        short s = 10;
        float f = 11.5F;

        ob.MiMétodo(i); // invoca ob.MiMétodo(int)
        ob.MiMétodo(d); // invoca ob.MiMétodo(double)

        ob.MiMétodo(b); // invoca ob.MiMétodo(byte) -- Ahora, no hay
conversión de tipo

        ob.MiMétodo(s); // invoca ob.MiMétodo(int) -- conversión de tipo
        ob.MiMétodo(f); // invoca ob.MiMétodo(double) -- conversión de tipo
    }
}
```

Ahora, cuando el programa se ejecuta, se producen los siguientes datos de salida:

```
Dentro de MiMétodo(int): 10
Dentro de MiMétodo(double): 10.1
Dentro de MiMétodo(byte): 99
Dentro de MiMétodo(int): 10
Dentro de MiMétodo(double): 11.5
```

Como en esta versión hay un **MiMétodo()** que recibe un argumento **byte**, cuando es invocado con dicho argumento se ejecuta **MiMétodo(byte)** y no ocurre la conversión automática a **int**.

Tanto **ref** como **out** intervienen en los propósitos de la recarga de métodos. En el siguiente ejemplo definen dos métodos distintos y separados:

```
public void MiMétodo(int x) {
    Console.WriteLine("Dentro de MiMétodo(int): " + x);
}
```

```
public void MiMétodo(ref int x) {  
    Console.WriteLine("Dentro de MiMétodo(ref int): " + x);  
}
```

De tal manera que,

```
ob.MiMétodo(i)
```

invoca **MiMétodo(int x)**, pero

```
ob.MiMétodo(ref i)
```

invoca **MiMétodo(ref int x)**.

Un punto de importancia: aunque **ref** y **out** intervienen en los propósitos de la recarga de métodos, la diferencia de cada uno de ellos por separado no es suficiente. Por lo mismo, las siguientes dos versiones de **MiMétodo()** son inválidas:

```
// Esto no se compilará.  
public void MiMétodo(out int x) { // ...  
public void MiMétodo(ref int x) { // ...
```

En este caso, el compilador no puede diferenciar entre las versiones de **MiMétodo()** simplemente porque una use **ref** y la otra **out**.

La recarga de métodos soporta polimorfismo porque es una manera en que C# implementa el paradigma “una interfaz, múltiples métodos”. Para entender cómo, considera sobre lo siguiente. En los lenguajes que no soportan la sobrecarga de métodos, a cada método debe dársele un nombre único. Sin embargo, con frecuencia querrás implementar esencialmente el mismo método para diferentes tipos de datos. Considera la función de valor absoluto. En los lenguajes que no soportan la sobrecarga, por lo regular hay tres o más versiones de la misma función, cada una con un nombre apenas diferente. Por ejemplo, en C, la función **abs()** regresa el valor absoluto de un entero, **labs()** regresa el valor absoluto de un entero largo y **fabs()** regresa el valor absoluto de un valor de punto flotante.

Como C no soporta la sobrecarga, cada una de las funciones de valor absoluto necesita tener su propio nombre, aunque las tres realicen en esencia la misma tarea. Esto complica conceptualmente la situación sin necesidad. Aunque el concepto subyacente de cada función es el mismo, aún tendrás que recordar tres nombres diferentes. Esto no ocurre en C# porque cada método de valor absoluto puede utilizar el mismo nombre. En efecto, la biblioteca de clases estándar de C# incluye un método de valor absoluto llamado **Abs()**. Este método es sobrecargado por la clase **System.Math** de C# para manejar los tipos numéricos. C# determina qué versión de **Abs()** invocará, dependiendo del tipo del argumento.

El valor de la sobrecarga es que permite acceder a métodos relacionados a través de un nombre común. Así, el nombre **Abs** representa la *acción general* que se está realizando. Al compilador se le delega la tarea de seleccionar la versión *específica* correcta para una situación en particular. Tú, el programador, sólo necesitas recordar la operación general. A través de la aplicación del polimorfismo, muchos nombres han sido reducidos a uno solo. Aunque este ejemplo es muy sencillo, si expandes el concepto, puedes ver cómo ayuda la sobrecarga a manejar situaciones de gran complejidad.

## Pregunta al experto

**P:** He escuchado el término *firma* entre los programadores de C#, ¿qué significa?

**R:** En lo que concierne a C#, una firma es el nombre de un método más su lista de parámetros. De esta manera, en lo concerniente a la sobrecarga, dos métodos dentro de la misma clase no podrán tener la misma firma. Observa que una firma no incluye el tipo de regreso, porque no es utilizado por C# para los propósitos de la sobrecarga.

Cuando sobrecargas un método, cada versión de él puede realizar cualquier actividad que le designes. No hay ninguna regla que establezca que los métodos sobrecargados deban estar relacionados entre sí. Sin embargo, desde el punto de vista del estilo, la sobrecarga de métodos implica una relación. De esta manera, aunque puedes utilizar el mismo nombre para sobrecargar métodos sin relación, no debes hacerlo. Por ejemplo, puedes utilizar el nombre **cuadrado** para crear métodos que regresen el *cuadrado* de un número entero y la *raíz cuadrada* de un valor de punto flotante, pero ambas operaciones son fundamentalmente diferentes. Aplicar la sobrecarga de métodos de esta manera abandona su propósito original. En la práctica sólo debes sobrecargar operaciones estrechamente relacionadas.

## Sobrecargar constructores

Como los métodos, los constructores también pueden ser sobrecargados. Hacerlo te permite construir objetos de muchas maneras. Como ejemplo, analiza el siguiente programa:

```
// Muestra un constructor sobrecargado.

using System;

class MiClase {
    public int x;

    public MiClase() { ←
        Console.WriteLine("Dentro de MiClase().");
        x = 0;
    }

    public MiClase(int i) { ←
        Console.WriteLine("Dentro de MiClase(int).");
        x = i;
    }

    public MiClase(double d) { ←
        Console.WriteLine("Dentro de MiClase(double).");
        x = (int)d;
    }
}
```

Construye un objeto de diversas formas.

```

public MiClase(int i, int j) { ← Construye un objeto
    Console.WriteLine("Dentro de MiClase(int, int).");
    x = i * j;
}
}

class SobrecargaConsDemo {
    static void Main() {
        MiClase t1 = new MiClase();
        MiClase t2 = new MiClase(88);
        MiClase t3 = new MiClase(17.23);
        MiClase t4 = new MiClase(2, 4);

        Console.WriteLine("t1.x: " + t1.x);
        Console.WriteLine("t2.x: " + t2.x);
        Console.WriteLine("t3.x: " + t3.x);
        Console.WriteLine("t4.x: " + t4.x);
    }
}

```

Los datos de salida generados por el programa son:

```

Dentro de MiClase().
Dentro de MiClase(int).
Dentro de MiClase(double).
Dentro de MiClase(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8

```

**MiClase()** se sobrecarga de cuatro maneras, cada una de las cuales construyen un objeto de modo diferente. El constructor apropiado se invoca con base en los argumentos especificados cuando se ejecuta **new**. Al sobrecargar un constructor de clase, le otorgas al usuario de la misma flexibilidad en la manera como los objetos están construidos.

Una de las razones más comunes para sobrecargar los constructores es permitir que un objeto inicialice otro. Por ejemplo, observa el siguiente programa que utiliza la clase **Sumatoria** para realizar esta operación sobre un valor entero:

```

// Inicializa un objeto con otro.

using System;

class Sumatoria {
    public int Sum;

    // Constructor de un entero.
    public Sumatoria(int num) {
        Sum = 0;
    }
}

```

```
for(int i=1; i <= num; i++)
    Sum += i;
}

// Constructor a partir de otro objeto.
public Sumatoria(Sumatoria ob) {← Construye un objeto a partir de otro.
    Sum = ob.Sum;
}

class SumDemo {
    static void Main() {
        Sumatoria s1 = new Sumatoria(5);
        Sumatoria s2 = new Sumatoria(s1);

        Console.WriteLine("s1.Sum: " + s1.Sum);
        Console.WriteLine("s2.Sum: " + s2.Sum);
    }
}
```

Los datos de salida son:

```
s1.Sum: 15
s2.Sum: 15
```

A menudo, como lo muestra el ejemplo anterior, la ventaja que proporciona un constructor que utiliza un objeto para inicializar otro es la eficiencia. En este caso, cuando **s2** es construido, no es necesario volver a calcular la sumatoria. Por supuesto, aun en casos donde la eficiencia no es lo importante, en muchas ocasiones es útil proporcionar un constructor que haga una copia de un objeto.

## Invocar un constructor sobrecargado a través de **this**

Cuando se trabaja con constructores sobrecargados, suele ser útil que un constructor invoque a otro. En C# esta tarea se consigue utilizando otro formato de la palabra clave **this**. El formato general se muestra a continuación:

```
nombre-constructor(lista-de-parámetros1) : this (lista-de-parámetros2) {
    // ... cuerpo del constructor, que puede estar vacío
}
```

Cuando el constructor se ejecuta, el constructor sobrecargado que coincide con la lista de parámetros especificada por *lista-de-parámetros2* es el primero en ejecutarse. Después, si hay algunas declaraciones dentro del constructor original, éstas se ejecutan. He aquí un ejemplo:

```
// Muestra la invocación de un constructor a través de this.

using System;

class CoordXY {
    public int x, y;
```

```

public CoordXY() : this(0, 0) {
    Console.WriteLine("Dentro de CoordXY()");
}

public CoordXY(CoordXY obj) : this(obj.x, obj.y) {
    Console.WriteLine("Dentro de CoordXY(CoordXY obj)");
}

public CoordXY(int i, int j) {
    Console.WriteLine("Dentro de CoordXY(CoordXY(int, int))");
    x = i;
    y = j;
}
}

class ConsSCDemo {
    static void Main() {
        CoordXY t1 = new CoordXY();
        CoordXY t2 = new CoordXY(8, 9);
        CoordXY t3 = new CoordXY(t2);

        Console.WriteLine("t1.x, t1.y: " + t1.x + ", " + t1.y);
        Console.WriteLine("t2.x, t2.y: " + t2.x + ", " + t2.y);
        Console.WriteLine("t3.x, t3.y: " + t3.x + ", " + t3.y);
    }
}

```

Usa **this** para invocar un constructor sobrecargado.

Los datos generados por el programa:

```

Dentro de CoordXY(CoordXY(int, int)
Dentro de CoordXY()
Dentro de CoordXY(CoordXY(int, int)
Dentro de CoordXY(CoordXY(int, int)
Dentro de CoordXY(CoordXY obj)
t1.x, t1.y: 0, 0
t2.x, t2.y: 8, 9
t3.x, t3.y: 8, 9

```

He aquí cómo funciona el programa. En la clase **CoordXY** el único constructor que realmente inicializa los campos **x** y **y** es **CoordXY(int, int)**. Los otros dos constructores simplemente invocan **CoordXY(int, int)** a través de **this**. Por ejemplo, cuando el objeto **t1** se crea, su constructor, **CoordXY()**, es invocado. Esto provoca que se ejecute **this(0, 0)**, el cual, en este caso, se traduce en una invocación a **CoordXY(0, 0)**.

Una de las razones por las que puede ser de utilidad invocar constructores sobrecargados a través de **this** es prevenir la duplicación innecesaria de código. En el ejemplo anterior, no hay ninguna razón por la que los tres constructores dupliquen la misma secuencia de inicialización, lo cual se evita utilizando **this**. Otra ventaja es que puedes crear constructores con “argumentos predeterminados” implícitos, los cuales son utilizados cuando estos argumentos no

son declarados explícitamente. Por ejemplo, podrías crear otro constructor **CoordXY**, como se muestra aquí:

```
public CoordXY(int x) : this (x, x) { }
```

Este constructor ajusta el valor de la coordenada **y** al mismo valor que la coordenada **x** predeterminadamente y en automático. Por supuesto, estos “argumentos predeterminados” deben utilizarse con cuidado porque su mal empleo puede confundir a los usuarios de tus clases.

**Prueba esto**

## Sobrecargar un constructor ColaSimple

En este ejemplo enriquecerás la clase **ColaSimple** desarrollada en secciones previas *Prueba esto*; ahora le darás dos constructores adicionales. El primero creará una nueva cola a partir de otra. El segundo construirá una cola, dándole valores iniciales. Como verás, añadir estos constructores enriquecerá el empleo de **ColaSimple** sustancialmente.

### Paso a paso

1. Crea un archivo llamado **QDemo2.cs**, y copia en ella la clase actualizada **ColaSimple** de la anterior sección *Prueba esto*.
2. Añade el siguiente constructor, que construye una cola a partir de otra.

```
// Construye una ColaSimple a partir de otra.
public ColaSimple(ColaSimple ob) {
    putloc = ob.putloc;
    getloc = ob.getloc;
    q = new char[ob.q.Length];

    // Copia elementos
    for(int i=getloc+1; i <= putloc; i++)
        q[i] = ob.q[i];
}
```

Observa con cuidado este constructor. Inicializa **putloc** y **getloc** con los valores contenidos en el parámetro **ob**. Después asigna un nuevo arreglo para contener la cola y copia los elementos de **ob** al arreglo. Una vez construido, la nueva cola será una copia idéntica del original, pero ambas serán objetos completamente independientes.

3. Añade el constructor que inicializa la cola a partir de un arreglo de caracteres, como se muestra aquí:

```
// Construye una ColaSimple con valores iniciales.
public ColaSimple(char[] a) {
    putloc = 0;
    getloc = 0;
    q = new char[a.Length+1];

    for(int i = 0; i < a.Length; i++) Put(a[i]);
}
```

Este constructor crea una cola lo bastante larga para contener los caracteres en **a** y luego almacena estos mismos caracteres en la cola. Dada la manera como funciona el algoritmo de la cola, la longitud de la misma debe ser 1 más grande que el arreglo.

- 4.** A continuación presentamos la clase **ColaSencilla** actualizada y completa, junto con la clase **QDemo2**, que la aplica:

```
// Una clase cola para caracteres.

using System;

class ColaSimple {

    // Ahora éstos son privados.
    char[] q; // este arreglo contiene la cola.
    int putloc, getloc; // los índices put y get.

    // Construye una ColaSencilla vacía dado su tamaño.
    public ColaSimple(int tamaño) {
        q = new char[tamaño+1]; // reserva memoria para la cola.
        putloc = getloc = 0;
    }

    // Construye una ColaSimple a partir de otra.
    public ColaSimple(ColaSimple ob) {
        putloc = ob.putloc;
        getloc = ob.getloc;
        q = new char[ob.q.Length];

        // Copia elementos.
        for(int i=getloc+1; i <= putloc; i++)
            q[i] = ob.q[i];
    }

    // Construye una ColaSimple con valores iniciales.
    public ColaSimple(char[] a) {
        putloc = 0;
        getloc = 0;
        q = new char[a.Length+1];

        for(int i = 0; i < a.Length; i++) Put(a[i]);
    }

    // Pone un carácter en la cola.
    public void Put(char ch) {
        if(putloc == q.Length-1) {
            Console.WriteLine(" -- La cola está llena.");
            return;
        }
    }
}
```

```
}

    putloc++;
    q[putloc] = ch;
}

// Obtiene un carácter de la cola.
public char Get() {
    if(getloc == putloc) {
        Console.WriteLine(" -- La cola está vacía.");
        return (char)0;
    }

    getloc++;
    return q[getloc];
}

// Muestra el funcionamiento de la clase ColaSimple.
class QDemo2 {
    static void Main() {
        // Construye una cola vacía de 10 elementos.
        ColaSimple q1 = new ColaSimple(10);

        char[] nombre = {'T', 'o', 'm'};

        // Construye una cola a partir de un arreglo.
        ColaSimple q2 = new ColaSimple(nombre);

        char ch;
        int i;

        // Coloca algunos caracteres en q1.
        for(i=0; i < 10; i++)
            q1.Put((char) ('A' + i));

        // Construye una cola a partir de otra.
        ColaSimple q3 = new ColaSimple(q1);

        // Muestra las colas.
        Console.Write("Contenido de q1: ");
        for(i=0; i < 10; i++) {
            ch = q1.Get();
            Console.Write(ch);
        }

        Console.WriteLine("\n");
```

```

Console.WriteLine("Contenido de q2: ");
for(i=0; i < 3; i++) {
    ch = q2.Get();
    Console.Write(ch);
}

Console.WriteLine("\n");

Console.WriteLine("Contenido de q3: ");
for(i=0; i < 10; i++) {
    ch = q3.Get();
    Console.Write(ch);
}
}
}
}

```

- 5.** Los datos de salida generados por el programa son:

Contenido de q1: ABCDEFGHIJ

Contenido de q2: Tom

Contenido de q3: ABCDEFGHIJ

## El método Main( )

Hasta este momento, has utilizado un solo formato de **Main( )**. Sin embargo, existen distintos formatos de sobrecarga para este método. Algunos pueden ser utilizados para regresar un valor y algunos otros pueden recibir un argumento. Cada uno es examinado a continuación.

### Regresar valores de Main( )

Cuando finaliza un programa, puedes regresar un valor al proceso invocador (por lo regular el sistema operativo) regresando un valor de **Main( )**. Para hacerlo, puedes utilizar este formato:

```
static int Main()
```

Observa que en lugar de declarar **void**, esta versión de **Main( )** tiene un regreso tipo **int**.

Por lo regular, el valor regresado por **Main( )** indica si el programa terminó de manera normal o debido a una condición extraordinaria. Por convención, un valor de regreso 0 suele indicar una terminación normal. Todos los demás valores indican que ocurrió algún tipo de error.

### Transmitir argumentos a Main( )

Muchos programas aceptan lo que recibe el nombre de argumentos de *línea de comando*. Un argumento de este tipo es la información que sigue directamente al nombre del programa en la línea de comandos cuando es ejecutado. Para los programas de C#, estos argumentos son transmitidos

entonces al método **Main( )**. Para recibir estos argumentos, debes utilizar uno de los siguientes formatos:

```
static void Main(string[ ] args)
```

```
static int Main(string[ ] args)
```

El primer formato regresa **void**, es decir, nada; el segundo puede ser utilizado para regresar un valor de entero, como se explicó en la sección anterior. En ambos casos, los argumentos de la línea de comandos son almacenados como cadenas de caracteres en el arreglo **string** transmitido a **Main( )**. La longitud del arreglo *args* será igual a la cantidad de argumentos de la línea de comandos.

Por ejemplo, el siguiente programa muestra todos los argumentos de la línea de comandos utilizados para invocarlo:

```
// Muestra toda la información de línea de comandos.

using System;

class CLDemo {
    static void Main(string[] args) {
        Console.WriteLine("Hay " + args.Length +
                          " argumentos en la línea de comandos.");
        Console.WriteLine("Que son: ");
        for(int i=0; i < args.Length; i++)
            Console.WriteLine(args[i]);
    }
}
```

Si **CLDemo** se ejecuta así:

```
CLDemo uno dos tres
```

Obtendrás los siguientes datos de salida:

```
Hay 3 argumentos en la línea de comandos.
Que son:
uno
dos
tres
```

Para ejemplificar la manera como pueden ser utilizados los argumentos de la línea de comandos, analiza el siguiente programa. Toma un argumento de la línea de comandos que especifica el nombre de una persona. Después busca a través de un arreglo bidimensional de cadenas de caracteres en busca del nombre. Si encuentra una coincidencia, muestra el número telefónico de esa persona.

```
// Un sencillo directorio telefónico automatizado.

using System;

class Teléfono {
    public static int Main(string[] args) {
        string[,] números = {
            { "Tom", "555-3322" },
            { "Mary", "555-8976" },
            { "Jon", "555-1037" },
            { "Rachel", "555-1400" },
        };
        int i;

        if(args.Length != 1) {
            Console.WriteLine("Uso: Teléfono <nombre>");
            return 1; // indica argumentos inadecuados de la línea de comandos.
        }
        else {
            for(i=0; i < números.Length/2; i++) {
                if(números[i, 0] == args[0]) {
                    Console.WriteLine(números[i, 0] + ": " +
                        números[i, 1]);
                    break;
                }
            }
            if(i == números.Length/2)
                Console.WriteLine("Nombre no encontrado.");
        }
        return 0;
    }
}
```

He aquí un ejemplo de ejecución:

```
C>Teléfono Mary
Mary: 555-8976
```

Hay dos elementos importantes en este programa. Primero, observa la manera en que el programa confirma que un argumento de la línea de comandos esté presente al verificar la longitud de **args**. Si el número de argumentos es incorrecto, la ejecución concluye. Esta confirmación es muy importante y puede generalizarse. Cuando un programa depende de la existencia de uno o más argumentos provenientes de la línea de comandos, siempre debe confirmar que han sido proporcionados los argumentos apropiados. Una falla en este aspecto ¡siempre generará una terminación abrupta del programa!

Segundo, observa cómo el programa regresa un código de finalización. Si la línea de comando requerida no está presente, entonces se regresa un 1, indicando una finalización anormal. De otra manera, se regresa un 0 cuando el programa termina normalmente.

## Recursión

En C# un método puede invocarse a sí mismo. Este proceso recibe el nombre de *recursión*, y al método que se invoca a sí mismo se le conoce como *recursivo*. En general la recursión es el proceso de definir algo en términos de sí mismo y de alguna manera es similar a una definición circular. El componente clave de un método recursivo es que contiene una declaración que ejecuta una invocación a sí mismo. La recursión es un mecanismo de control muy poderoso.

El ejemplo clásico de la recursión es el cálculo del factorial de un número. El factorial de un número  $N$  es el producto de la multiplicación de todos los números entre 1 y  $N$ . Por ejemplo, el factorial de 3 es  $1 \times 2 \times 3$ , o 6. El siguiente programa muestra una manera recursiva de calcular el factorial de un número. Para efectos de comparación, también se incluye un equivalente no recursivo.

```
// Un ejemplo sencillo de recursión.

using System;

class Factorial {
    // Ésta es una función recursiva.
    public int FactR(int n) {
        if(n==1) return 1;
        else return FactR(n-1) * n; ← Ejecuta una invocación recursiva a FactR().
    }

    // Éste es su equivalente repetitivo.
    public int FactI(int n) {
        int t, result;

        result = 1;
        for(t=1; t <= n; t++) result *= t;
        return result;
    }
}

class Recursion {
    static void Main() {
        Factorial f = new Factorial();

        Console.WriteLine("Factoriales utilizando el método recursivo.");
        Console.WriteLine("Factorial de 3 es " + f.FactR(3));
        Console.WriteLine("Factorial de 4 es " + f.FactR(4));
        Console.WriteLine("Factorial de 5 es " + f.FactR(5));
        Console.WriteLine();

        Console.WriteLine("Factoriales utilizando el método de repetición.");
        Console.WriteLine("Factorial de 3 es " + f.FactI(3));
        Console.WriteLine("Factorial de 4 es " + f.FactI(4));
        Console.WriteLine("Factorial de 5 es " + f.FactI(5));
    }
}
```

Los datos de salida generados por el programa son los siguientes:

Factoriales utilizando el método recursivo.

Factorial de 3 es 6

Factorial de 4 es 24

Factorial de 5 es 120

Factoriales utilizando el método de repetición.

Factorial de 3 es 6

Factorial de 4 es 24

Factorial de 5 es 120

La operación del método no recursivo **FactI()** debe ser clara. Utiliza un loop comenzando en 1 y multiplica cada número por el producto resultante.

La operación del método recursivo **FactR()** es un poco más compleja. Cuando se invoca **FactR()** con un argumento de 1, el método regresa un 1; de otra manera, regresa el producto de **FactR(n-1)\*n**. Para evaluar esta expresión, **FactR()** es invocado con **n-1**. Este proceso se repite hasta que **n** es igual a 1 y las invocaciones al método comienzan a regresar. Por ejemplo, cuando se calcula el factorial de 2, la primera invocación a **FactR()** provocará que sea hecha una segunda invocación con 1 como argumento. Esta invocación regresará 1, que luego se multiplica por 2 (el valor original de **n**). Entonces la respuesta es 2. Tal vez te resulte interesante insertar declaraciones **WriteLine()** en **FactR()** para mostrar en qué nivel se encuentra cada invocación y cuál es el resultado intermedio.

Cuando un método se invoca a sí mismo, se les asigna almacenamiento a nuevas variables locales y parámetros en la pila del sistema, y el código del método se ejecuta con estas nuevas variables desde el principio. (Una invocación recursiva *NO* hace una nueva copia del método.) Conforme cada invocación recursiva regresa, las variables locales y los parámetros antiguos se eliminan de la pila y la ejecución reinicia en el punto de invocación dentro del método. Se puede decir que los métodos recursivos “salen y entran” con información.

Es posible que la versión recursiva de muchas rutinas se ejecute un poco más lento en comparación con sus equivalentes reiterativos, debido a la sobrecarga adicional de invocaciones. Muchas invocaciones recursivas a un método pueden provocar que la pila se desborde. Dado que el almacenamiento de parámetros y variables locales se encuentra en la pila y cada nueva invocación crea una nueva copia de estas variables, es posible que la pila se agote. Si esto ocurre, el CLR lanzará una excepción. Sin embargo, es muy probable que no tengas que preocuparte de este asunto a menos que una rutina recursiva se salga de control.

La principal ventaja de la recursión es que algunas clases de algoritmos pueden ser implementados con mayor claridad y sencillez de manera recursiva, en comparación con la reiteración. Por ejemplo, el algoritmo de ordenamiento rápido es muy difícil de implementar de manera reiterativa. Además, algunos problemas, en especial los relacionados con AI, parecen tender naturalmente a una solución recursiva.

Cuando escribes métodos recursivos, debes tener una declaración condicional en alguna parte, como un **if**, con el fin de forzar el regreso del método sin que se ejecute la invocación recursiva. Si no lo haces, una vez que invoques el método, éste nunca regresará. Este tipo de error es muy común cuando se trabaja con la recursión. Utiliza **WriteLine()** libremente para que observes lo que va sucediendo y abortar la ejecución en caso de que detectes un error.

## Comprender static

Habrá ocasiones en las que quieras definir un miembro de clase que se utilice independientemente de cualquier objeto de esa clase. Por lo regular, un miembro de clase debe ser accesado por un objeto de su clase, pero es posible crear un miembro que sólo pueda ser utilizado por sí mismo, sin hacer referencia a una instancia específica. Para crear un miembro así, antecede su declaración con la palabra clave **static**. Cuando un miembro es declarado **static**, puede accesarse antes de que sea creado cualquier objeto de su clase y sin hacer referencia a otro objeto. Puedes declarar **static** tanto métodos como variables. El ejemplo más común de un miembro **static** es **Main()**, que es declarado **static** porque debe ser invocado por el sistema operativo cuando inicia tu programa.

Fuera de la clase, para utilizar un miembro **static** debes especificar el nombre de su clase seguido por el operador de punto. No es necesario que se cree ningún objeto. De hecho, un miembro **static** no puede ser accesado a través de una instancia de objeto. Debe ser accesado a través del nombre de su clase. Por ejemplo, supongamos que una variable **static** llamada **cuenta** es miembro de una clase llamada **Tiempo**. Para asignar a **cuenta** el valor 10 se utilizaría la siguiente línea:

```
Tiempo.cuenta = 10;
```

Este formato es similar al utilizado para accesar variables de instancia normales a través de un objeto, sólo que en este caso se utiliza el nombre de la clase. Un método **static** puede ser invocado de la misma manera, utilizando el operador de punto en el nombre de la clase.

Las variables que se declaran **static** son, en esencia, variables globales. Cuando se declaran objetos de su clase, no se hacen copias de las variables **static**. En vez de ello, todas las instancias de la clase comparten la misma variable **static**. Una variable **static** es inicializada antes de que su clase se utilice. Si no se especifica un inicializador específico, comienza con un valor igual a cero en los tipos numéricos, nulo para los tipos de referencia o **false** para las variables de tipo **bool**. De esta manera, las variables **static** siempre tienen un valor.

La diferencia entre un método **static** y uno normal es que el primero puede ser invocado a través de su nombre de clase, sin necesidad de que se cree una instancia de esa clase. Ya has visto un ejemplo de esto: el método **Sqrt()**, que es **static** y está definido dentro de la clase **System.Math** de C#.

A continuación presentamos un ejemplo de declaraciones **static** en variables y métodos:

```
// Uso de static.

using System;

class StaticDemo {

    // Una variable static.
    public static int Val = 100; ← Una variable static.

    // Un método static.
    public static int ValDiv2() { ← Un método static.
        return Val/2;
    }
}
```

```

class SDemo {
    static void Main() {

        Console.WriteLine("El valor inicial de StaticDemo.Val es "
            + StaticDemo.Val);

        StaticDemo.Val = 8;
        Console.WriteLine("StaticDemo.Val es " + StaticDemo.Val);
        Console.WriteLine("StaticDemo.ValDiv2(): " +
            StaticDemo.ValDiv2());
    }
}

```

Los datos de salida generados por el programa son:

```

El valor inicial de StaticDemo.Val es 100
StaticDemo.Val es 8
StaticDemo.ValDiv2(): 4

```

Como lo demuestran los datos de salida, una variable **static** se inicializa antes de que se cree cualquier objeto de su clase.

Existen varias restricciones que se aplican a los métodos **static**:

- Un método **static** no tiene una referencia **this**. Lo cual se debe a que un método de este tipo no se ejecuta con relación a ningún objeto.
- Un método **static** sólo puede invocar directamente otros métodos **static** de su misma clase. No puede invocar directamente un método de instancia de su propia clase. La razón es que los métodos de instancia operan sobre objetos específicos, pero un método **static** no es invocado sobre un objeto. Así pues, ¿sobre qué objeto operaría el método **instancia**?
- Una restricción similar aplica a los datos **static**. Un método **static** puede accesar directamente sólo otros datos **static** de su clase. No puede operar sobre una variable de instancia de su clase porque no hay objeto sobre el cual operar.

Por ejemplo, en la siguiente clase, el método **static ValDivDenom()** es ilegal:

```

class StaticError {
    public int Denom = 3; // una variable de instancia normal
    public static int Val = 1024; // una variable static

    /* ¡Error! No es posible accesar una variable diferente a static
       desde el interior de un método static. */
    public static int ValDivDenom() {
        return Val/Denom; // ¡No se compilará!
    }
}

```

Aquí, **Denom** es una variable de instancia normal que no puede ser accesada dentro de un método **static**. Sin embargo, el uso de **Val** sí es permitido, porque es una variable **static**.

El mismo problema ocurre cuando se intenta invocar un método normal desde el interior de un método **static** de la misma clase. Por ejemplo:

```
using System;

class OtroErrorStatic {

    // Un método no static.
    public void MetNoStatic() {
        Console.WriteLine("Dentro de MetNoStatic().");
    }

    /* ¡Error! No es posible invocar un método no static
       desde el interior de un método static. */
    public static void StaticMeth() {
        MetNoStatic(); // No compilará
    }
}
```

En este caso, el intento de invocar un método no **static** (es decir, de instancia) desde un método **static** causa un error en tiempo de compilación.

Es importante entender que aunque un método **static** no puede invocar directamente métodos de instancia ni accesar variables de instancia de su clase, *sí puede* invocar un método de instancia o accesar una variable de instancia si lo hace a través de un objeto de su clase. Simplemente no puede utilizar una variable o método de instancia de manera directa, sin la habilitación de un objeto. Por ejemplo, el siguiente fragmento es perfectamente válido:

```
class MiClase {
    // Un método no static.
    public void MetNoStatic() {
        Console.WriteLine("Dentro de MetNoStatic().");
    }

    /* Es posible invocar un método no static a través de
       una referencia de objeto desde el interior de un método static. */
    public static void StaticMeth(MiClase ob) {
        ob.MetNoStatic(); // esto es válido
    }
}
```

## Constructores y clases static

Existen otros dos usos de **static** que por lo regular se aplican en situaciones más avanzadas que las descritas en este capítulo. Sin embargo, con fines de exhaustividad, aquí se describen brevemente.

Además de variables y métodos, es posible especificar un constructor como **static**. Por ejemplo,

```
class Ejemplo {
    // ...
    static Ejemplo() { ... }
```

Aquí, **Ejemplo()** se declara **static** y es, por lo mismo, un constructor **static** para la clase **Ejemplo**. Un constructor **static** es invocado automáticamente cuando la clase se carga por primera vez, antes de la construcción de objetos y antes de la invocación de cualquier constructor de instancia. Así, el uso primario de un constructor **static** es inicializar características que se aplicarán a la clase como un todo, en lugar de a una sola instancia de la clase. Un constructor **static** no puede tener modificadores de acceso y no puede ser invocado directamente por tu programa. Más aún, tiene las mismas restricciones que los métodos **static**, descritas anteriormente.

Desde la versión 2.0 de C#, también puedes especificar una clase como **static**. Por ejemplo:

```
static class Test { // ... }
```

Una clase **static** tiene dos características importantes. La primera, no es posible crear ningún objeto de la clase **static**. La segunda, una clase de este tipo sólo puede tener miembros **static**. El uso principal de una clase **static** se encuentra cuando se trabaja con métodos de extensión, que es una característica avanzada añadida a C# 3.0.

## Prueba esto

## Acomodo rápido

En el capítulo 5 se mostró un método sencillo para ordenar datos llamado *bubble*. También se mencionó que hay mejores maneras de realizar esa tarea. Aquí desarrollarás una de las mejores: el acomodo rápido. El acomodo rápido (**Quicksort**), inventado y llamado así por C.A.R. Hoare, es el mejor algoritmo de acomodo para propósitos generales que actualmente existe. La razón por la que no podía ser explicado en el capítulo 5 es que sus mejores implementaciones descansan en la recursión. De tal manera que es un excelente ejemplo para mostrar el poder de la recursión en acción. La versión que desarrollaremos acomoda un arreglo de caracteres, pero la lógica puede ser aplicada para acomodar cualquier tipo de objetos que elijas.

El acomodo rápido está construido sobre la idea de *particiones*. El procedimiento general es seleccionar un valor, llamado *comparando*, y luego partir el arreglo en dos secciones. Todos los elementos mayores que o iguales al valor de la partición son colocados en un lado, y los que son menores a dicho valor en el otro. El proceso se repite para cada sección restante hasta que el arreglo se termina de acomodar. Por ejemplo, dado el arreglo **fedacb** y utilizando **d** como comparando, el primer pase de acomodo rápido ordenaría el arreglo de la siguiente manera:

Inicial	f e d a c b
Pase 1	b c a d e f

Este proceso se repite para cada sección, es decir, para **bca** y **def**. Como puedes ver, se trata de un proceso esencialmente recursivo en su naturaleza y, en efecto, la implementación más limpia del acomodo rápido es por un método recursivo.

(continúa)

Puedes seleccionar el comparando de dos maneras. Puedes hacer una selección aleatoria, o bien puedes seleccionarlo al promediar un pequeño conjunto de valores tomados del arreglo. Para un acomodo óptimo debes seleccionar un valor que esté precisamente en medio del rango de valores. Sin embargo, esta tarea es difícil de realizar en la mayoría de los conjuntos de datos. En el peor de los casos, el elemento seleccionado puede encontrarse en un extremo. Pero aun en este caso, el acomodo rápido se ejecuta correctamente. La versión de acomodo rápido que desarrollaremos selecciona el elemento medio del arreglo como comparando.

Un punto más antes de comenzar. Aunque desarrollar tus propios métodos de acomodo es instructivo y divertido, por lo regular no tendrás que hacerlo. La razón es que C# proporciona métodos de biblioteca que acomodian arreglos y otras colecciones de objetos. Por ejemplo, el método **System.Array.Sort()** puede acomodar cualquier arreglo. Como era de esperarse, ¡este método también se implementa como un acomodo rápido!

## Paso a paso

1. Crea un archivo llamado **QSDemo.cs**.
2. Crea la clase **Quicksort** (acomodo rápido) que se muestra a continuación:

```
// Una versión sencilla de acomodo rápido.

using System;

class Quicksort {

    // Establece una invocación al método actual de acomodo rápido.
    public static void QSort(char[] items) {
        if(items.Length == 0) return;
        qs(items, 0, items.Length-1);
    }

    // Una versión recursiva de acomodo rápido para caracteres.
    private static void qs(char[] items, int left, int right)
    {
        int i, j;
        char x, y;

        i = left; j = right;
        x = items[(left+right)/2];

        do {
            while((items[i] < x) && (i < right)) i++;
            while((x < items[j]) && (j > left)) j--;

            if(i <= j) {
                y = items[i];
                items[i] = items[j];
                items[j] = y;
                i++; j--;
            }
        } while (i <= j);
    }
}
```

```

        if(left < j) qs(items, left, j);
        if(i < right) qs(items, i, right);
    }
}

```

Para mantener sencilla la interfaz del acomodo rápido, la clase **Quicksort** proporciona el método **QSort( )**, que establece una invocación al verdadero método de acomodo rápido, **qs( )**. Esto permite que el arreglo rápido sea invocado con sólo el nombre del arreglo que será ordenado, sin necesidad de proporcionar una partición inicial. Como **qs( )** sólo se ocupa internamente, se especifica como **private**.

- 3.** Para utilizar **Quicksort** simplemente se invoca así: **Quicksort.QSort( )**. Como **QSort( )** se especifica como **static**, debe ser invocada a través de su clase en lugar de hacerlo por un objeto. Así las cosas, no hay necesidad de crear un objeto **Quicksort**. Después de que regrese la invocación, el arreglo será ordenado. Recuerda, esta versión sólo funciona para arreglos de caracteres, pero puedes adaptar la lógica para acomodar cualquier tipo de arreglo que deseas.
- 4.** Aquí está el programa que muestra la función de **Quicksort**:

```

// Una versión sencilla de acomodo rápido.

using System;

class Quicksort {

    // Establece una invocación al método actual de acomodo rápido.
    public static void QSort(char[] items) {
        if(items.Length == 0) return;
        qs(items, 0, items.Length-1);
    }

    // Una versión recursiva de acomodo rápido para caracteres.
    private static void qs(char[] items, int left, int right)
    {
        int i, j;
        char x, y;

        i = left; j = right;
        x = items[(left + right)/2];

        do {
            while((items[i] < x) && (i < right)) i++;
            while((x < items[j]) && (j > left)) j--;

            if(i <= j) {
                y = items[i];
                items[i] = items[j];
                items[j] = y;
                i++; j--;
            }
        }
    }
}

```

```
        }
    } while(i <= j);

    if(left < j) qs(items, left, j);
    if(i < right) qs(items, i, right);
}
}

class QSDemo {
    static void Main() {
        char[] a = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
        int i;

        Console.WriteLine("Arreglo original: ");
        for(i=0; i < a.Length; i++) Console.Write(a[i]);

        Console.WriteLine();

        // Ahora, ordena el arreglo.
        Quicksort.QSort(a);

        Console.WriteLine("Arreglo acomodado: ");
        for(i=0; i < a.Length; i++) Console.Write(a[i]);
    }
}
```

## ✓ Autoexamen Capítulo 6

1. Dado este fragmento:

```
class X {
    int cuenta;
```

¿es correcto el siguiente fragmento?

```
class Y {
    static void Main() {
        X ob = new X();
        ob.cuenta = 10;
```

2. Un especificador de acceso debe \_\_\_\_\_ una declaración de miembro.

3. El complemento de una cola es una pila. Utiliza el tipo de acceso: “el primero que entra es el último que sale” y por lo regular se compara con una pila de platos. El primer plato colocado sobre la mesa es el último que se usa. Crea una clase de pila llamada **Pila** que contenga caracteres. Nombra los métodos que accesan la pila **Empujar()** y **Meter()**. Permite que el usuario

especifique el tamaño de la pila cuando ésta es creada. Mantén al resto de los miembros de la clase **Pila** como **private**. Consejo: puedes utilizar la clase **ColaSimple** como modelo; simplemente cambia la manera como se accesan los datos.

- 4.** Dada esta clase:

```
class Test {  
    int a;  
    Test(int i) { a = i; }  
}
```

escribe un método llamado **Permuta()** que intercambie el contenido de los objetos a los que se refieren dos referencias de objeto **Test**.

- 5.** ¿Es correcto el siguiente fragmento?

```
class X {  
    int meth(int a, int b) { ... }  
    string meth (int a, int b) { ... }
```

- 6.** Escribe un método recursivo que invierta y muestre el contenido de una cadena de caracteres.  
**7.** Si todos los objetos de una clase necesitan compartir la misma variable, ¿cómo debes declarar esa variable?  
**8.** ¿Qué hacen **ref** y **out**? ¿En qué se diferencian?  
**9.** Muestra cuatro formatos de **Main()**.  
**10.** Dado el siguiente fragmento, ¿cuáles de las posteriores invocaciones son legales?

```
void meth(int i, int j, params int [] args) { // ...
```

- A.** meth(10, 12, 19);
- B.** meth(10, 12, 19, 100);
- C.** meth(10, 12, 19, 100, 200);
- D.** meth(10, 12);



# Capítulo 7

Sobrecarga de  
operadores,  
indexadores y  
propiedades

## Habilidades y conceptos clave

- Fundamentos de la sobrecarga de operadores
  - Sobrecargar operadores binarios
  - Sobrecargar operadores unitarios
  - Sobrecargar operadores relacionales
  - Indexadores
  - Propiedades
- 

Este capítulo examina tres clases especiales de miembros de clase: operadores sobrecargados, indexadores y propiedades. Cada uno de ellos expande el poder de una clase mejorando su utilidad, su integración en el sistema de tipos de C# y su flexibilidad. Al utilizar estos miembros, es posible crear tipos de clase con la misma apariencia y sensación de los tipos integrados. Esta *extensibilidad de tipos* es una parte importante del poder de un lenguaje orientado a objetos como C#.

## Sobrecarga de operadores

C# te permite definir el significado de un operador con relación a la clase que creas. Esta operación lleva el nombre de *sobrecarga de operadores*. Al sobrecargar un operador expandes su uso a tu clase. Los efectos del operador están completamente bajo tu control y son diferentes de clase a clase. Por ejemplo, una clase que define una lista vinculada puede utilizar el operador + para añadir un objeto a la lista. Una clase que implementa una pila, puede ocupar el operador + para empujar un objeto en la misma. Otra clase puede utilizar el operador + de una manera completamente diferente.

Cuando un operador es sobrecargado, no se pierde ninguno de sus significados originales. Simplemente se añade una nueva operación con relación a la clase específica donde se aplica. Por lo mismo, sobrecargar el operador + para manejar una lista vinculada, por ejemplo, no provoca que cambie su significado relativo a los enteros (esto es, la función de suma).

Una de las ventajas principales de la sobrecarga de operadores es que te permite integrar un nuevo tipo de clase sin fisuras al ambiente de tu programa. Una vez que los operadores para una clase están definidos, puedes operar sobre objetos de esa clase utilizando las expresiones sintácticas normales de C#. Incluso puedes utilizar un objeto en expresiones que involucren otros tipos de datos.

La sobrecarga de operadores está estrechamente vinculada con la sobrecarga de métodos. Para sobrecargar un operador utiliza la palabra clave **operator** para definir un *método operador*, que a su vez define la acción del operador.

## Los formatos generales de un método operador

Existen dos formatos para los métodos **operator**: uno para operadores unitarios y otro para binarios. El formato general de cada uno se muestra a continuación:

```
// Formato general para sobrecargar un operador unitario.
public static tipo-ret operador op(tipo-param operando)
{
    // operaciones
}
```

```
// Formato general para sobrecargar un operador binario.
public static tipo-ret operador op(tipo1-param operando1, tipo2-param operando2)
{
    // operaciones
}
```

Aquí, el operador que estás sobrecargando, como `+` o `/`, está representado por `op`. El *tipo-ret* es el tipo de valor returnedo por la operación especificada. Aunque puede ser de cualquier tipo que elijas, por lo regular el valor que regresa es del mismo tipo que la clase para la cual el operador está siendo sobrecargado. Esta correlación facilita el uso del operador sobrecargado en expresiones. Para operadores unitarios, el operando es transmitido en la parte del formato que lleva el mismo nombre (*operando*). Para operadores binarios, los operandos son transmitidos en *operando1* y *operando2*.

Para operadores unitarios, el operando debe ser el mismo tipo de la clase para la cual está siendo definido. Para operadores binarios, al menos uno de los operandos debe ser del mismo tipo de la clase. Así, no puedes sobrecargar ningún operador de C# para objetos que no hayas creado. Por ejemplo, no puedes redefinir `+` para **int** o **string**.

Otro punto: los parámetros de los operadores no deben utilizar los modificadores **ref** ni **out**.

## Sobrecargar operadores binarios

Para ver cómo funciona la sobrecarga de operadores, comenzemos con un ejemplo que sobrecarga dos operadores binarios: el `+` y el `-`. El siguiente programa crea una clase llamada **TresD**, que mantiene las coordenadas de un objeto en un espacio tridimensional. El operador `+` sobrecargado suma las coordenadas individuales del objeto **TresD** a otro. El operador `-` sobrecargado resta las coordenadas de un objeto a otro.

```
// Un ejemplo de sobrecarga de operadores.

using System;

// Una clase de coordenadas tridimensionales.
class TresD {
    int x, y, z; // coordenadas 3-D

    public TresD() { x = y = z = 0; }
    public TresD(int i, int j, int k) { x = i; y = j; z = k; }
```

```
// Sobre carga el operador binario +.
public static TresD operator +(TresD op1, TresD op2)
{
    TresD result = new TresD();      Sobrecarga + para objetos TresD.
    ↑

    /* Esto suma las coordenadas de dos puntos
       y regresa el resultado. */
    result.x = op1.x + op2.x; // Éstas son sumas de enteros
    result.y = op1.y + op2.y; // y el + retiene su significado
    result.z = op1.z + op2.z; // original con relación a ellas.

    return result;
}

// Sobre carga del operador binario -.
public static TresD operator -(TresD op1, TresD op2)
{
    TresD result = new TresD();      Sobrecarga - para objetos TresD.
    ↑

    /* Observa el orden de los operandos. op1 es el operando
       izquierdo y op2 es el derecho. */
    result.x = op1.x - op2.x; // éstas son restas de enteros.
    result.y = op1.y - op2.y;
    result.z = op1.z - op2.z;

    return result;
}

// Muestra las coordenadas de X, Y, Z.
public void Show()
{
    Console.WriteLine(x + ", " + y + ", " + z);
}
}

class TresDDemo {
    static void Main()
    {
        TresD a = new TresD(1, 2, 3);
        TresD b = new TresD(10, 10, 10);
        TresD c = new TresD();

        Console.Write("Aquí está a: ");
        a.Show();
        Console.WriteLine();
        Console.Write("Aquí está b: ");
        b.Show();
        Console.WriteLine();
```

```

c = a + b; // suma a y b juntas ←
Console.WriteLine("Resultado de a + b: ");
c.Show();
Console.WriteLine();

c = a + b + c; // suma a, b y c juntas ←
Console.WriteLine("Resultado de a + b + c: ");
c.Show();
Console.WriteLine();

c = c - a; // resta a ←
Console.WriteLine("Resultado de c - a: ");
c.Show();
Console.WriteLine();

c = c - b; // resta b ←
Console.WriteLine("Resultado de c - b: ");
c.Show();
Console.WriteLine();
}

}

```

Utiliza objetos **TresD** y el + y el - en las expresiones.

El programa genera los siguientes datos de salida:

```

Aquí está a: 1, 2, 3
Aquí está b: 10, 10, 10
Resultado de a + b: 11, 12, 13
Resultado de a + b + c: 22, 24, 26
Resultado de c - a: 21, 22, 23
Resultado de c - b: 11, 12, 13

```

Examinemos con cuidado este programa, comenzando con el operador sobrecargado `+`. Cuando dos objetos del tipo **TresD** son operados por `+`, las magnitudes de sus respectivas coordenadas se suman, como se muestra en `operator+( )`. Observa, sin embargo, que este método no modifica el valor de ningún operando. En vez de eso, el método regresa un nuevo objeto del tipo **TresD**, que contiene el resultado de la operación. Para comprender por qué la operación `+` no cambia el contenido de ningún objeto, piensa en la operación aritmética estándar de la suma (`+`) aplicada así:  $10 + 12$ . El resultado de esta operación es 22, pero ni el 10 ni el 12 son modificados por la operación misma. Aunque no hay ninguna regla que impida que un operador sobrecargado altere el valor de uno de sus operandos, lo mejor es que sea consistente con su significado habitual.

Observa que `operator+( )` regresa un objeto de tipo **TresD**. Aunque el método podría regresar cualquier tipo válido C#, el hecho de que regrese un objeto **TresD** permite que el operador `+` sea

utilizado en expresiones combinadas, como **a+b+c**. Aquí, **a+b** generan un resultado que es de tipo **TresD**. Luego, este valor puede ser sumado a **c**. Si **a+b** hubiesen generado cualquier otro tipo de valor, esa expresión no funcionaría.

He aquí otro punto de importancia: cuando las coordenadas se suman dentro de **operator+( )**, la suma de las coordenadas individuales resulta en una adición de enteros. Esto se debe a que las coordenadas **x**, **y** y **z** son cantidades enteras. El hecho de que el operador **+** esté sobrecargado para objetos tipo **TresD** no tiene efecto sobre **+** cuando éste se aplica a valores enteros.

Ahora, observa el **operator-( )**. El operador **-** funciona justo como lo hace el operador **+**, salvo que el orden de los parámetros es importante. Recuerda que la suma es conmutativa, pero no la resta. (Es decir, **A-B** ¡no es lo mismo que **B-A!**) Para todos los operadores binarios, el primer parámetro para un método operador contendrá el operando izquierdo. El segundo parámetro contendrá el operando de la derecha. Cuando se implementan versiones sobrecargadas de operaciones no conmutativas, debes recordar qué operando está a la izquierda y cuál está a la derecha.

## Sobrecargar operadores unitarios

Los operadores unitarios se sobrecargan de la misma manera que los binarios. La principal diferencia, por supuesto, es que sólo existe un operando. Por ejemplo, he aquí un método que sobrecarga el menos unitario para la clase **TresD**:

```
// Sobre cargar el - unitario.
public static TresD operator -(TresD op)
{
    TresD result = new TresD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
}
```

Aquí se crea un nuevo objeto que contiene los campos negados del operando. Luego, este objeto es regresado. Advierte que el operando no sufre ningún cambio. De nuevo, esto se hace para mantener concordancia con el significado habitual de la resta unitaria. Por ejemplo, en una expresión como

**a = -b**

**a** recibe la negación de **b**, pero **b** no se modifica.

En C#, sobrecargar **++** y **--** es muy sencillo; simplemente regresa el valor incrementado o reducido, pero no modifica el objeto invocado. C# manejará eso automáticamente por ti, tomando en consideración la diferencia entre los formatos de prefijo y sufijo. Por ejemplo, a continuación presentamos un método **operator++( )** para la clase **TresD**:

```
// ++ unitario sobre cargar.
public static TresD operator ++(TresD op)
{
    TresD result = new TresD();
```

```
// Regresa el resultado incrementado.  
result.x = op.x + 1;  
result.y = op.y + 1;  
result.z = op.z + 1;  
  
return result;  
}
```

He aquí una versión expandida del programa anterior que muestra el funcionamiento del operador de sustracción – y el de adición ++:

```
// Más operadores sobrecargados.  
  
using System;  
  
// Una clase de coordenadas tridimensionales.  
class TresD {  
    int x, y, z; // Coordenadas 3-D  
  
    public TresD() { x = y = z = 0; }  
    public TresD(int i, int j, int k) { x = i; y = j; z = k; }  
  
    // Sobrecarga operador binario +.  
    public static TresD operator +(TresD op1, TresD op2)  
    {  
        TresD result = new TresD();  
  
        /* Esto suma las coordenadas de los dos puntos  
         y regresa el resultado. */  
        result.x = op1.x + op2.x; // Éstas son sumas de enteros  
        result.y = op1.y + op2.y; // y el + conserva su significado  
        result.z = op1.z + op2.z; // original relativo a ellas.  
  
        return result;  
    }  
  
    // Sobrecarga del operador binario -.  
    public static TresD operator -(TresD op1, TresD op2)  
    {  
        TresD result = new TresD();  
  
        /* Observa el orden de los operandos. op1 es el operando  
         izquierdo y op2 es el derecho. */  
        result.x = op1.x - op2.x; // éstas son restas de enteros  
        result.y = op1.y - op2.y;  
        result.z = op1.z - op2.z;  
  
        return result;  
    }
```

```
// Sobre carga - unitario.
public static TresD operator -(TresD op) ← Implementa menos unitario para TresD.
{
    TresD result = new TresD();

    result.x = -op.x;
    result.y = -op.y;
    result.z = -op.z;

    return result;
}

// Sobre carga ++ unitario.
public static TresD operator ++(TresD op) ← Implementa ++ para TresD.
{
    TresD result = new TresD();

    // Regresa el resultado incrementado.
    result.x = op.x + 1;
    result.y = op.y + 1;
    result.z = op.z + 1;

    return result;
}

// Muestra coordenadas X, Y, Z.
public void Show()
{
    Console.WriteLine(x + ", " + y + ", " + z);
}
}

class TresDDemo {
    static void Main() {
        TresD a = new TresD(1, 2, 3);
        TresD b = new TresD(10, 10, 10);
        TresD c = new TresD();

        Console.Write("Aquí está a: ");
        a.Show();
        Console.WriteLine();
        Console.Write("Aquí está b: ");
        b.Show();
        Console.WriteLine();

        c = a + b; // suma a y b
        Console.Write("Resultado de a + b: ");
        c.Show();
        Console.WriteLine();
```

```
c = a + b + c; // suma a, b y c
Console.WriteLine("Resultado de a + b + c: ");
c.Show();
Console.WriteLine();

c = c - a; // resta a
Console.WriteLine("Resultado de c - a: ");
c.Show();
Console.WriteLine();

c = c - b; // resta b
Console.WriteLine("Resultado de c - b: ");
c.Show();
Console.WriteLine();

c = -a; // asigna -a a c
Console.WriteLine("Resultado de -a: ");
c.Show();
Console.WriteLine();

c = a++; // incremento posterior a
Console.WriteLine("Dado c = a++");
Console.WriteLine("c es ");
c.Show();
Console.WriteLine("a es ");
a.Show();

// Restablece a a 1, 2, 3
a = new TresD(1, 2, 3);
Console.WriteLine("\nRestablece a a ");
a.Show();

c = ++a; // incremento previo a
Console.WriteLine("\nDado c = ++a");
Console.WriteLine("c es ");
c.Show();
Console.WriteLine("a es ");
a.Show();
}
}
```

Los datos generados por el programa son:

```
Aquí está a: 1, 2, 3
Aquí está b: 10, 10, 10
Resultado de a + b: 11, 12, 13
```

Resultado de a + b + c: 22, 24, 26

Resultado de c - a: 21, 22, 23

Resultado de c - b: 11, 12, 13

Resultado de -a: -1, -2, -3

Dado c = a++  
c es 1, 2, 3  
a es 2, 3, 4

Restablece a a 1, 2, 3

Dado c = ++a  
c es 2, 3, 4  
a es 2, 3, 4

## Añadir flexibilidad

Para cualquier clase y operador dados, un método operador puede, por sí mismo, ser sobrecargado. Por ejemplo, una vez más considera la clase **TresD**. En este punto, has visto cómo sobrecargar el operador **+** con el fin de que sume las coordenadas de un objeto **TresD** a otro. Sin embargo, ésta no es la única manera como quisiéramos definir la suma para **TresD**. Por ejemplo, sería útil sumar un valor entero a cada coordenada de un objeto **TresD**. Una operación así serviría para mover los ejes. Para realizar esta operación necesitarías sobrecargar por segunda ocasión el operador **+**, como se muestra aquí:

```
// Sobre cargar el operador binario + para TresD + int.  
public static TresD operator +(TresD op1, int op2)  
{  
    TresD result = new TresD();  
  
    result.x = op1.x + op2;  
    result.y = op1.y + op2;  
    result.z = op1.z + op2;  
  
    return result;  
}
```

Observa que el segundo parámetro es de tipo **int**. Así, el método anterior permite que se sume un valor entero a cada campo de un objeto **TresD**. Esto es posible porque, como se examinó anteriormente, cuando se sobre carga un operador binario, uno de los operandos tiene que ser del mismo tipo de la clase para la cual el operador está siendo sobre cargado. Sin embargo, el otro operando puede ser de cualquier otro tipo.

He aquí una versión de **TresD** que tiene dos métodos + sobrecargados:

```
// Sobrecarga suma para TresD + TresD, y para TresD + int.

using System;

// Una clase de coordenadas tridimensionales.
class TresD {
    int x, y, z; // Coordenadas 3-D

    public TresD() { x = y = z = 0; }
    public TresD(int i, int j, int k) { x = i; y = j; z = k; }

    // Sobrecarga operador binario + para TresD + TresD.
    public static TresD operator +(TresD op1, TresD op2)
    {
        TresD result = new TresD();

        /* Esto suma las coordenadas de dos puntos
         * y regresa el resultado. */
        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;

        return result;
    }

    // Sobrecarga operador binario + para TresD + int.
    public static TresD operator +(TresD op1, int op2)
    {
        ↑
        TresD result = new TresD();           Define TresD + entero.

        result.x = op1.x + op2;
        result.y = op1.y + op2;
        result.z = op1.z + op2;

        return result;
    }

    // Muestra coordenadas X, Y, Z.
    public void Show()
    {
        Console.WriteLine(x + ", " + y + ", " + z);
    }
}
```

```
class TresDDemo {
    static void Main() {
        TresD a = new TresD(1, 2, 3);
        TresD b = new TresD(10, 10, 10);
        TresD c = new TresD();

        Console.WriteLine("Aquí está a: ");
        a.Show();
        Console.WriteLine();
        Console.WriteLine("Aquí está b: ");
        b.Show();
        Console.WriteLine();

        c = a + b; // TresD + TresD
        Console.WriteLine("Resultado de a + b: ");
        c.Show();
        Console.WriteLine();

        c = b + 10; // TresD + int
        Console.WriteLine("Resultado de b + 10: ");
        c.Show();
    }
}
```

Los datos de salida generados por el programa son:

```
Aquí está a: 1, 2, 3
Aquí está b: 10, 10, 10
Resultado de a + b: 11, 12, 13
Resultado de b + 10: 20, 20, 20
```

Como lo confirma el resultado, cuando se aplica `+` a dos objetos **TresD**, sus coordenadas se suman. Cuando `+` se aplica a un objeto **TresD** y a un entero, sus coordenadas se incrementan de acuerdo con el valor del entero.

Como lo acaba de demostrar la sobrecarga de `+`, añadir útiles capacidades a la clase **TresD** no concluye por completo la tarea. He aquí la razón. El método **operator+(TresD, int)** permite declaraciones como la siguiente:

```
obj1 = obj2 + 10;
```

Desafortunadamente no permite declaraciones como ésta:

```
obj1 = 10 + obj2;
```

porque el argumento entero es el segundo, que es el operando de la derecha. El problema es que el argumento anterior pone el entero a la izquierda. Para permitir ambas formas de declaración, nece-

sitarás sobrecargar de nuevo el operador `+`. Esta versión debe tener el primer parámetro como tipo `int` y el segundo como tipo `TresD`. Una versión del método `operator+( )` maneja `TresD + entero`, y el otro maneja `entero + TresD`. Sobrecargar de esta manera el operador `+` (o cualquier otro operador binario) permite que ocurra un tipo integrado en cualquiera de los dos lados de la operación, izquierdo o derecho. He aquí una versión de `TresD` que sobrecarga el operador `+` como el que acabamos de describir:

```
// Sobrecarga de + para TresD + TresD, TresD + int e int + TresD.

using System;

// Una clase de coordenadas tridimensionales.
class TresD {
    int x, y, z; // Coordenadas 3-D

    public TresD() { x = y = z = 0; }
    public TresD(int i, int j, int k) { x = i; y = j; z = k; }

    // Sobrecarga operador binario + para TresD + TresD.
    public static TresD operator +(TresD op1, TresD op2)
    {
        TresD result = new TresD();           Esto maneja TresD + TresD.

        /* Esto suma las coordenadas de dos puntos
           y regresa el resultado. */
        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;

        return result;
    }

    // Sobrecarga operador binario + para TresD + int.
    public static TresD operator +(TresD op1, int op2)
    {
        TresD result = new TresD();           Esto maneja TresD + entero.

        result.x = op1.x + op2;
        result.y = op1.y + op2;
        result.z = op1.z + op2;

        return result;
    }

    // Sobrecarga operador binario + para int + TresD.
    public static TresD operator +(int op1, TresD op2)
    {
        TresD result = new TresD();           Esto maneja entero + TresD.
```

```
        result.x = op2.x + op1;
        result.y = op2.y + op1;
        result.z = op2.z + op1;

        return result;
    }

    // Muestra coordenadas X, Y, Z.
    public void Show()
    {
        Console.WriteLine(x + " , " + y + " , " + z);
    }
}

class TresDDemo {
    static void Main()
    {
        TresD a = new TresD(1, 2, 3);
        TresD b = new TresD(10, 10, 10);
        TresD c = new TresD();

        Console.Write("Aquí está a: ");
        a.Show();
        Console.WriteLine();
        Console.Write("Aquí está b: ");
        b.Show();
        Console.WriteLine();

        c = a + b; // TresD + TresD
        Console.Write("Resultado de a + b: ");
        c.Show();
        Console.WriteLine();

        c = b + 10; // TresD + int
        Console.Write("Resultado de b + 10: ");
        c.Show();
        Console.WriteLine();

        c = 15 + b; // int + TresD
        Console.Write("Resultado de 15 + b: ");
        c.Show();
    }
}
```

El programa genera los siguientes datos de salida:

```
Aquí está a: 1, 2, 3
Aquí está b: 10, 10, 10
```

Resultado de a + b: 11, 12, 13

Resultado de b + 10: 20, 20, 20

Resultado de 15 + b: 25, 25, 25

## Sobrecargar los operadores de relación

Los operadores de relación, como == o <, también pueden ser sobrecargados, y el proceso es sencillo. Por lo regular, un operador relacional sobrecargado regresa un valor **true** o **false**. Esto es así para mantener el uso normal de estos operadores y permitir que los operadores relacionales sobrecargados sean utilizados en expresiones condicionales. Si regresas un resultado con un tipo diferente, restringes de manera considerable la utilidad del operador.

He aquí una versión de la clase **TresD** que sobrecarga los operadores < y >. En este ejemplo, los operadores comparan objetos **TresD** basándose en su distancia a partir del origen. Un objeto es más grande que otro si su distancia a partir del origen es superior. Un objeto es menor que otro, si la distancia a partir del origen es menor que la del otro. Dados dos puntos, una implementación así podría ser utilizada para determinar cuál punto se encuentra en una esfera superior. Si ningún operador regresa **true**, los dos puntos se encuentran en la misma esfera. Por supuesto, es posible aplicar otros esquemas de organización.

```
// Sobrecarga < y >.

using System;

// Una clase de coordenadas tridimensional.
class TresD {
    int x, y, z; // Coordenadas 3-D

    public TresD() { x = y = z = 0; }
    public TresD(int i, int j, int k) { x = i; y = j; z = k; }

    // Sobrecarga <.
    public static bool operator <(TresD op1, TresD op2)
    {
        if (Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) <
            Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
            return true;
        else
            return false;
    }

    // Sobrecarga >.
    public static bool operator >(TresD op1, TresD op2)
    {
        if (Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z) >
            Math.Sqrt(op2.x * op2.x + op2.y * op2.y + op2.z * op2.z))
            return true;
    }
}
```

Regresa true si **op1** es menor que **op2**.

↓

Regresa true si **op1** es mayor que **op2**.

↓

```
        else
            return false;
    }

    // Muestra coordenadas X, Y, Z.
    public void Show()
    {
        Console.WriteLine(x + " , " + y + " , " + z);
    }
}

class TresDDemo {
    static void Main()
    {
        TresD a = new TresD(5, 6, 7);
        TresD b = new TresD(10, 10, 10);
        TresD c = new TresD(1, 2, 3);
        TresD d = new TresD(6, 7, 5);

        Console.Write("Aquí está a: ");
        a.Show();
        Console.Write("Aquí está b: ");
        b.Show();
        Console.Write("Aquí está c: ");
        c.Show();
        Console.Write("Aquí está d: ");
        d.Show();
        Console.WriteLine();

        if (a > c) Console.WriteLine("a > c es true");
        if (a < c) Console.WriteLine("a < c es true");
        if (a > b) Console.WriteLine("a > b es true");
        if (a < b) Console.WriteLine("a < b es true");

        if (a > d) Console.WriteLine("a > d es true");
        else if (a < d) Console.WriteLine("a < d es true");
        else Console.WriteLine("a y d están a la misma distancia del
        origen");
    }
}
```

Los datos de salida generados por este programa son los siguientes:

```
Aquí está a: 5, 6, 7
Aquí está b: 10, 10, 10
Aquí está c: 1, 2, 3
Aquí está d: 6, 7, 5

a > c es true
a < b es true
a y d están a la misma distancia del origen
```

Hay una restricción importante que se aplica a la sobrecarga de operadores relacionales: debes sobrecargarlos en pares. Por ejemplo, si sobrecargas `<`, también tienes que sobrecargar `>`, y viceversa. Las parejas de operadores son las siguientes:

<code>==</code>	<code>!=</code>
<code>&lt;</code>	<code>&gt;</code>
<code>&lt;=</code>	<code>&gt;=</code>

Así, si sobrecargas `<=`, también tienes que sobrecargar `>=`, y si sobrecargas `==`, también tienes que sobrecargar `!=`.

### NOTA

Si sobrecargas los operadores `==` y `!=`, por lo general tendrás que anular `Object.Equals()` y `Object.GetHashCode()`. Estos métodos y la técnica de anulación se estudian en el capítulo 8.

## Consejos y restricciones para la sobrecarga de operadores

La acción de un operador sobrecargado como se aplica a la clase para la cual fue definido no necesita tener una relación con el uso predeterminado de ese mismo operador, tal y como lo especifica el tipo integrado de C#. Sin embargo, para propósitos de estructura y legibilidad del código, un operador sobrecargado debe reflejar, en la medida de lo posible, el espíritu de su uso original. Por ejemplo, el `+` relativo a `TresD` es conceptualmente similar al `+` relativo a los tipos enteros. Sería muy poco benéfico definir el operador `+` relativo a una clase para que actuara como uno esperaría que lo hiciera el operador `/`. El concepto central es que puedes hacer que un operador sobrecargado tenga el significado que deseas, pero por claridad es mejor que ese nuevo significado esté relacionado con el uso original del operador.

Existen algunas restricciones para sobrecargar operadores. No puedes alterar la precedencia de ningún operador. No puedes alterar el número de operandos requeridos por el operador, aunque tu método de operador pueda elegir ignorar un operando. Existen varios operadores que no pueden ser sobrecargados. Tal vez lo más importante es que no puedes sobrecargar ningún operador de asignación, incluyendo las asignaciones de combinación, como `+=`. He aquí los operadores que no pueden ser sobrecargados. Algunos de ellos serán estudiados más tarde en este mismo libro.

<code>&amp;&amp;</code>	<code>()</code>	<code>.</code>	<code>?</code>
<code>??</code>	<code>[]</code>	<code>  </code>	<code>=</code>
<code>=&gt;</code>	<code>-&gt;</code>	<code>as</code>	<code>checked</code>
<code>default</code>	<code>is</code>	<code>new</code>	<code>sizeof</code>
<code>typeof</code>	<code>unchecked</code>		

Un último punto: las palabras clave `true` y `false` también pueden ser utilizadas como operadores unitarios para propósitos de sobrecarga. Se sobrecargan con relación a la clase que determina si un objeto es “verdadero” o “falso”. Una vez que han sido sobrecargadas para una clase, puedes utilizar objetos de esa clase para controlar una declaración `if`, por ejemplo.

## Pregunta al experto

**P:** Cómo no puedo sobrecargar operadores como `+=`, ¿qué sucede si intento utilizar `+=` con un objeto de una clase para la cual he definido `+`, por ejemplo? De manera más general, ¿qué sucede cuando utilizo cualquier asignación de combinación sobre un objeto para el cual he definido una parte operacional de esa asignación?

**R:** En general, si has definido un operador, cuando éste es utilizado en una asignación de combinación, tu método de operador sobrecargado es invocado. Así, `+=` automáticamente utiliza tu versión de `operator+( )`. Por ejemplo, retomando la clase `TresD`, si utilizas una secuencia como ésta:

```
TresD a = new TresD(1, 2, 3);
TresD b = new TresD(10, 10, 10);

b += a; // suma a y b
```

Se invoca automáticamente el `operator+( )` de `TresD`, y `b` contendrá las coordenadas 11, 12 y 13.

## Indexadores

Como sabes, la indexación de arreglos se realiza utilizando el operador `[ ]`. Es posible sobrecargar éste para las clases que tú crees, pero no se utiliza el método `operator`. En lugar de ello, creas un *indexador*. Un indexador permite que un objeto sea indexado como un arreglo. El uso principal de los indexadores es soportar la creación de arreglos especializados que son sujetos de una o más restricciones. Sin embargo, puedes utilizar los indexadores para cualquier propósito donde la sintaxis de los arreglos te proporcione algún beneficio. Los indexadores pueden tener una o más dimensiones.

Comenzaremos con indexadores de una dimensión. Los indexadores unidimensionales tienen el siguiente formato general:

```
tipo-elemento this[int índice] {
    // El accesador get
    get {
        // regresa los valores especificados por índice
    }
}

// El accesador set.
set {
    // establece el valor especificado por índice
}
```

Aquí, *tipo-elemento* es, precisamente, el tipo de elemento del indexador. Así, cada elemento accesado por el indexador será del *tipo-elemento*. Se corresponde al tipo de elemento de un arreglo. El parámetro *índice* recibe el índice del elemento que será accesado. Técnicamente, este elemento no tiene que ser de tipo **int**, pero como los indexadores son utilizados generalmente para proporcionar una indexación de los arreglos, un tipo entero es lo habitual.

Dentro del cuerpo del indexador se definen dos *accesadores*, llamados **get** y **set**. Un accesador es similar a un método, excepto que no declara un tipo de regreso ni un parámetro. Los accesadores son invocados automáticamente cuando se utiliza el indexador y ambos reciben el *índice* como parámetro. Si el indexador ha sido asignado, como cuando se encuentra del lado izquierdo en una declaración de asignación, entonces el accesador **set** es invocado y el elemento especificado por *índice* debe ser establecido. De lo contrario, se invoca el accesador **get** y el valor asociado con el *índice* debe ser regresado. El método **set** también recibe un valor llamado **value**, que contiene el valor asignado al *índice* especificado.

Uno de los beneficios del indexador es que puedes controlar con precisión la manera como se accesa un arreglo, desechando accesos inapropiados. Por ejemplo, a continuación presentamos una mejor manera de implementar el arreglo “falla leve” creado en el capítulo 6. Esta versión utiliza un indexador, permitiendo así que el arreglo sea accesado utilizando la notación estándar.

```
// Mejora el arreglo de falla leve añadiendo un indexador.

using System;

class ArregloFallaLeve {
    int[] a;      // referencia al arreglo

    public int Length; // Longitud es public

    public bool ErrFlag; // indica los datos de salida de la última
    operación

    // Construye el arreglo dependiendo de su tamaño.
    public ArregloFallaLeve(int tamaño) {
        a = new int[tamaño];
        Length = tamaño;
    }

    // Éste es el indexador para ArregloFallaLeve.
    public int this[int index] { ←———— Un indexador para ArregloFallaLeve.
        // Éste es el accesador get.
        get {
            if (ok(index)) {
                ErrFlag = false;
                return a[index];
            } else {
                ErrFlag = true;
                return 0;
            }
        }
    }
}
```

```

// Éste es el accesador set.
set {
    if (ok(index)) {
        a[index] = value;
        ErrFlag = false;
    }
    else ErrFlag = true;
}

// Regresa true si el índice está dentro del límite.
private bool ok(int index) {
    if (index >= 0 & index < Length) return true;
    return false;
}

// Muestra el arreglo de falla leve mejorado.
class MejoraFSDemo {
    static void Main() {
        ArregloFallaLeve fs = new ArregloFallaLeve(5);
        int x;

        // Muestra las fallas silenciosas.
        Console.WriteLine("Falla silenciosa.");
        for (int i=0; i < (fs.Length * 2); i++)
            fs[i] = i*10; ← Invoca al accesador set del indexador.

        for (int i=0; i < (fs.Length * 2); i++) {
            x = fs[i]; ← Invoca al accesador get del indexador.
            if (x != -1) Console.Write(x + " ");
        }
        Console.WriteLine();

        // Ahora, genera fallas.
        Console.WriteLine("\nFalla con reporte de errores.");
        for (int i=0; i < (fs.Length * 2); i++) {
            fs[i] = i*10;
            if (fs.ErrFlag)
                Console.WriteLine("fs[" + i + "] fuera de límites");
        }

        for (int i=0; i < (fs.Length * 2); i++) {
            x = fs[i];
            if (!fs.ErrFlag) Console.Write(x + " ");
            else
                Console.WriteLine("fs[" + i + "] fuera de límites");
        }
    }
}

```

Los datos de salida generados por el programa son:

```
Falla silenciosa.
0 10 20 30 40 0 0 0 0 0

Falla con reportes de error.
fs[5] fuera de límites
fs[6] fuera de límites
fs[7] fuera de límites
fs[8] fuera de límites
fs[9] fuera de límites
0 10 20 30 40 fs[5] fuera de límites
fs[6] fuera de límites
fs[7] fuera de límites
fs[8] fuera de límites
fs[9] fuera de límites
```

Los datos de salida son similares a los producidos por la versión previa del programa mostrado en el capítulo 6. En esta versión, el indexador previene que los límites del arreglo se desborden. Veamos de cerca cada parte del indexador. Comienza con esta línea:

```
public int this [int index] {
```

Esto declara un indexador que opera sobre elementos **int**. El índice es transmitido en **index**. El indexador es público, lo que le permite ser utilizado por código fuera de su clase.

El accesador **get** se muestra aquí:

```
get {
    if (ok(index)) {
        ErrFlag = false;
        return a[index];
    } else {
        ErrFlag = true;
        return 0;
    }
}
```

El accesador **get** previene errores de límite del arreglo. Si el índice especificado está dentro de los límites, el elemento correspondiente al índice es regresado. Si está fuera de los límites, no se realiza ninguna operación ni tampoco ocurre un desbordamiento. En esta versión de **ArregloFalladoLeve**, una variable llamada **ErrFlag** contiene el resultado de cada operación. Este campo puede ser examinado después de cada operación para evaluar el éxito o fracaso de la misma. (En el capítulo 10, verás mejores maneras de manejar los errores utilizando el sistema de excepciones de C#, por el momento, utilizar una advertencia de error como **ErrFlag** es suficiente.)

El accesador **set** se muestra aquí; también previene un error de límites.

```
set {
    if (ok(index)) {
        a[index] = value;
```

```
    ErrFlag = false;
}
else ErrFlag = true;
}
```

En este caso, si **index** está dentro de los límites, el valor transmitido en **value** es transmitido al elemento correspondiente. De lo contrario, **ErrFlag** se establece como **true**. Recuerda que dentro de un método accesador, **value** es un parámetro automático que contiene el valor que está siendo asignado. No necesitas (ni puedes) declararlo.

No es necesario que un indexador proporcione **get** ni **set**. Puedes crear un indexador sólo-lectura implementando únicamente el accesador **get**. Puedes crear un indexador sólo-escritura implementando únicamente **set**.

Es importante comprender que no hay requerimientos para que un indexador opere sobre un arreglo verdadero. Simplemente debe proporcionar funcionalidad que parezca “como un arreglo” al usuario del indexador. Por ejemplo, el siguiente programa tiene un indexador que actúa como un arreglo sólo-lectura que contiene las potencias de 2 del 0 al 15. Observa, sin embargo, que no existe ningún arreglo. En lugar de ello, el indexador simplemente calcula el valor adecuado para el índice dado.

```
// Los indexadores no tienen que actuar sobre arreglos verdaderos.

using System;

class PotDeDos {

    /* Accesa un arreglo lógico que contiene
       las potencias de 2 del 0 al 15. */
    public int this[int index] {
        // Calcula y regresa potencias de 2.
        get {
            if ((index >= 0) && (index < 16)) return Pwr(index);
            else return -1;
        }
        // No hay accesador set.
    }

    int Pwr(int p) {
        int result = 1;

        for (int i=0; i < p; i++)
            result *= 2;

        return result;
    }
}

class UsaPotDeDos {
    static void Main() {
        PotDeDos pwr = new PotDeDos();
```

Aquí no se utiliza un arreglo subyacente.

```

Console.WriteLine("Primeras 8 potencias de 2: ");
for (int i=0; i < 8; i++)
    Console.Write(pwr[i] + " ");
Console.WriteLine();

Console.WriteLine("Aquí hay algunos errores: ");
Console.WriteLine(pwr[-1] + " " + pwr[17]);
}
}

```

Los datos de salida generados por el programa son los siguientes:

```

Primeras 8 potencias de 2: 1 2 4 8 16 32 64 128
Aquí hay algunos errores: -1 -1

```

Observa que el indexador de **PotDeDos** incluye un accesador **get**, pero carece de un accesador **set**. Como ya explicamos, esto significa que se trata de un accesador sólo-lectura. Así, un objeto **PotDeDos** puede ser utilizado sobre el lado derecho de una declaración de asignación, pero no en el lado izquierdo. Por ejemplo, intentar añadir la siguiente declaración al programa anterior generaría un error:

```
pwr[0] = 11; // no compilará.
```

Esta declaración causaría un error de compilación porque no hay un accesador **set** definido por el indexador.

## Indexadores multidimensionales

También puedes crear indexadores para arreglos multidimensionales. Por ejemplo, aquí hay un arreglo bidimensional de falla leve. Observa con mucha atención la manera como se declara el indexador.

```

// Un arreglo bidimensional de falla leve.

using System;

class ArregloFallaLeve2D {
    int[,] a; // referencia al arreglo 2D
    int rows, cols; // dimensiones
    public int Length; // Length es público

    public bool ErrFlag; // indica el resultado de la última operación

    // Construye el arreglo según sus dimensiones.
    public ArregloFallaLeve2D(int r, int c) {
        rows = r;
        cols = c;
        a = new int[rows, cols];
        Length = rows * cols;
    }
}

```

```
// Éste es el indexador para ArregloFallaLeve2D.
public int this[int index1, int index2] { ← Un indexador de dos dimensiones.
    // Éste es el accesador get.
    get {
        if(ok(index1, index2)) {
            ErrFlag = false;
            return a[index1, index2];
        } else {
            ErrFlag = true;
            return 0;
        }
    }

    // Éste es el accesador set.
    set {
        if(ok(index1, index2)) {
            a[index1, index2] = value;
            ErrFlag = false;
        }
        else ErrFlag = true;
    }
}

// Regresa true si los índices están dentro de los límites.
private bool ok(int index1, int index2) {
    if(index1 >= 0 & index1 < rows &
       index2 >= 0 & index2 < cols)
        return true;

    return false;
}
}

// Muestra un indexador 2D.
class DosDIndexadorDemo {
    static void Main() {
        ArregloFallaLeve2D fs = new ArregloFallaLeve2D(3, 5);
        int x;

        // Muestra fallas silenciosas.
        Console.WriteLine("Falla silenciosa.");
        for (int i=0; i < 6; i++)
            fs[i, i] = i*10;

        for(int i=0; i < 6; i++) {
            x = fs[i, i];
            if(x != -1) Console.Write(x + " ");
        }
    }
}
```

```
Console.WriteLine();

// Ahora, genera fallas.
Console.WriteLine("\nFalla con reportes de error.");
for (int i=0; i < 6; i++) {
    fs[i,i] = i*10;
    if (fs.ErrFlag)
        Console.WriteLine("fs[" + i + ", " + i + "] fuera de límites");
}

for (int i=0; i < 6; i++) {
    x = fs[i,i];
    if(!fs.ErrFlag) Console.Write(x + " ");
    else
        Console.WriteLine("fs[" + i + ", " + i + "] fuera de límites");
}
}
```

Los datos de salida generados por el programa son:

```
Falla silenciosa.

0 10 20 0 0 0

Falla con reportes de error.
fs[3, 3] fuera de límites
fs[4, 4] fuera de límites
fs[5, 5] fuera de límites
0 10 20 fs[3, 3] fuera de límites
fs[4, 4] fuera de límites
fs[5, 5] fuera de límites
```

## Restricciones para los indexadores

Hay dos restricciones importantes para utilizar indexadores. Primera, dado que un indexador no define una localización de almacenamiento, el valor producido por él no puede ser transmitido como parámetros **ref** ni **out** a un método. Segunda, un indexador no puede ser declarado **static**.

### Pregunta al experto

**P:** ¿Se pueden sobrecargar los indexadores?

**R:** Sí. La versión que se ejecuta será la que tenga la coincidencia de tipo más cercana entre los parámetros y argumentos utilizados como índice.

## Propiedades

Otro tipo de miembro de clase es la *propiedad*. Como regla general, una propiedad combina un campo con métodos que lo accesan. Como lo han mostrado ejemplos anteriores de este libro, generalmente querrás crear un campo que esté disponible para los usuarios de un objeto, pero deseas mantener control sobre las operaciones autorizadas para ese campo. Por ejemplo, puedes querer limitar el rango de valores que le son asignados a ese campo. Es posible realizar esta tarea a través del uso de una variable privada, junto con métodos que tengan acceso a su valor, pero la propiedad ofrece una propuesta más racional.

Las propiedades son similares a los indexadores. Una propiedad está conformada por un nombre y accesadores **get** y **set**. Los accesadores se utilizan para recuperar (**get**) y establecer (**set**) el valor de la variable. El beneficio clave de la propiedad consiste en que su nombre puede ser utilizado en expresiones y asignaciones como una variable normal, pero en realidad los accesadores **get** y **set** son invocados automáticamente. Esto es similar a la manera en que **get** y **set** son utilizados en automático por los indexadores.

El formato general de una propiedad se muestra a continuación:

```
tipo nombre {
    get {
        // recupera código del accesador
    }

    set {
        // establece código del accesador
    }
}
```

Aquí, *tipo* especifica el tipo de la propiedad, como **int**, y *nombre* es el nombre de la propiedad. Una vez que la propiedad ha sido definida, cualquier uso del *nombre* trae como resultado una invocación a su accesador apropiado. El accesador **set** automáticamente recibe un parámetro llamado **value** que contiene el valor asignado a la propiedad.

Las propiedades no definen una locación de almacenamiento. En vez de ello, regularmente una propiedad maneja el acceso a un campo definido en alguna otra parte. La propiedad en sí no proporciona este campo. Así, el campo debe ser especificado independientemente de la propiedad. (La excepción es la propiedad *autoimplementada*, añadida en la versión 3.0 de C#, que será descrita en breve.)

He aquí un ejemplo sencillo que define una propiedad llamada **MiProp**, que es utilizada para accesar el campo **prop**. En este caso, la propiedad sólo permite la asignación de valores positivos.

```
// Un ejemplo sencillo de propiedad.

using System;

class SimpProp {
    int prop; // campo manejado por MiProp

    public SimpProp() { prop = 0; }
```

```

/* Ésta es la propiedad que soporta el acceso a la variable
   de instancia privada prop. Sólo permite valores positivos.*/
public int MiProp { ← Una propiedad llamada MiProp
    get {                               que controla el acceso a prop.
        return prop;
    }
    set {
        if(value >= 0) prop = value;
    }
}

// Muestra una propiedad.
class PropiedadDemo {
    static void Main() {
        SimpProp ob = new SimpProp();

        Console.WriteLine("Valor original de ob.MiProp: " + ob.MiProp);

        ob.MiProp = 100; // asigna valor
        Console.WriteLine("Valor de ob.MiProp: " + ob.MiProp); ←
        // No es posible asignar valor negativo a prop.
        Console.WriteLine("Intenta asignar -10 a ob.MiProp");
        ob.MiProp = -10;
        Console.WriteLine("Valor de ob.MiProp: " + ob.MiProp);
    }
}

```

Utiliza **MiProp** como una variable.

Los datos generados por el programa son:

```

Valor original de ob.MiProp: 0
Valor de ob.MiProp: 100
Intenta asignar -10 a ob.MiProp
Valor de ob.MiProp: 100

```

Examinemos este programa con cuidado. El programa define un campo privado, llamado **prop**, y una propiedad llamada **MiProp** que maneja el acceso a **prop**. Como se explicó anteriormente, una propiedad en sí no define una locación de almacenamiento. En lugar de ello, la mayoría de las propiedades simplemente manejan el acceso a un campo. Más aún, como **prop** es privado, *sólo* puede ser accedido a través de **MiProp**.

La propiedad **MiProp** se especifica como **public**, por lo que se puede acceder por código fuera de su clase. Esto tiene sentido, pues proporciona acceso a **prop**, que es privado. El accesador **get** simplemente regresa el valor de **prop**. El accesador **set** establece el valor de **prop** si y sólo si ese valor es positivo. De esta manera, la propiedad **MiProp** controla los valores que puede tener **prop**. Ésta es la razón fundamental de la importancia de las propiedades.

El tipo de propiedad definido por **MiProp** es llamado propiedad lectura-escritura porque permite que su campo subyacente sea leído y escrito. No obstante, es posible crear propiedades

sólo-lectura y propiedades sólo-escritura. Para crear una propiedad sólo-lectura, define sólo un accesador **get**. Para definir una propiedad sólo-escritura, define sólo un accesador **set**.

Puedes utilizar una propiedad para mejorar más la clase arreglo de falla leve. Como sabes, todos los arreglos tienen una propiedad **Length** asociada a ellos. Hasta ahora, la clase **ArregloFallalLeve** simplemente usa un campo de entero público llamado **Length** para este propósito. Pero ésta no es una buena práctica, porque permite que **Length** sea establecida con un valor diferente a la longitud del arreglo de falla leve. (Por ejemplo, un programador malicioso puede intentar corromper este valor.) Podemos remediar esta situación transformando **Length** en una propiedad sólo-lectura, como se muestra en esta versión de **ArregloFallalLeve**:

```
// Añade propiedad Length a ArregloFallalLeve.

using System;

class ArregloFallalLeve {
    int[] a; // referencia al arreglo
    int len; // longitud del arreglo

    public bool ErrFlag; // indica el resultado de la última operación.

    // Construye el arreglo de acuerdo con su tamaño.
    public ArregloFallalLeve(int tamaño) {
        a = new int[tamaño];
        len = tamaño;
    }

    // Propiedad Length sólo-lectura.
    public int Length { ← Ahora Length es una propiedad y no un campo.
        get {
            return len;
        }
    }

    // Éste es el indexador para ArregloFallalLeve.
    public int this[int index] {
        // Éste es el accesador get.
        get {
            if (ok(index)) {
                ErrFlag = false;
                return a[index];
            } else {
                ErrFlag = true;
                return 0;
            }
        }

        // Éste es el accesador set.
        set {
            if (ok(index)) {
                a[index] = value;
            }
        }
    }
}
```

```

        ErrFlag = false;
    }
    else ErrFlag = true;
}
}

// Regresa true si el índice está dentro de los límites.
private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
}

// Muestra el arreglo de falla leve mejorado.
class MejorFSDemo {
    static void Main() {
        ArregloFallaLeve fs = new ArregloFallaLeve(5);
        int x;

        // No puede leer Length.
        for(int i=0; i < (fs.Length); i++)
            fs[i] = i*10;

        for(int i=0; i < (fs.Length); i++) {
            x = fs[i];
            if(x != -1) Console.Write(x + " ");
        }
        Console.WriteLine();
    }
}

```

Ahora **Length** es una propiedad sólo-lectura que puede ser leída pero no puede recibir ninguna asignación. Para que lo compruebes, intenta eliminar el signo de comentario que antecede esta línea del programa:

```
// fs.Length = 10; // ;Error, intento ilegal!
```

Cuando intentes compilar el programa recibirás un mensaje de error que dirá que **Length** es sólo-lectura.

## Propiedades autoimplementadas

Comenzando con C# 3.0, es posible implementar propiedades muy sencillas sin necesidad de definir explícitamente la variable manejada por la propiedad. En lugar de ello, puedes dejar que el compilador aporte la variable subyacente de manera automática. Esto lleva el nombre de *propiedad autoimplementada*. Tiene el siguiente formato general:

```
tipo nombre { get; set; }
```

Aquí, *tipo* especifica el tipo de la propiedad y *nombre* especifica su nombre. Observa que **get** y **set** van seguidos inmediatamente por un punto y coma. Los accesadores para una propiedad autoimplementada no tienen cuerpo. Esta sintaxis le indica al compilador que cree automáticamente una locación de almacenamiento (algunas veces llamada *campo trasero*) para contener el valor. Esta variable no es nombrada ni está directamente disponible para ti. En lugar de ello, sólo se puede acceder a través de la propiedad.

He aquí cómo se declara una propiedad llamada **CuentaUsuario** utilizando una propiedad autoimplementada:

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

Observa que no está declarada explícitamente ninguna variable. Como se explicó, el compilador genera de manera automática un campo anónimo que contiene el valor. En caso contrario, **CuentaUsuario** actúa como cualquier otra propiedad y como tal es utilizada.

Contrariamente a las propiedades normales, una propiedad autoimplementada no puede ser sólo-lectura ni sólo-escritura. En todos los casos se debe especificar tanto **get** como **set**. Sin embargo, puedes obtener un resultado semejante al declarar **get** o **set** como **private**, como se explicó hace poco.

Aunque las propiedades autoimplementadas ofrecen algunas ventajas, su uso está limitado a aquellos casos en los que no necesitas control sobre la recuperación o el establecimiento del campo trasero. Recuerda que no puedes accesar directamente el campo trasero. Esto significa que no hay ninguna manera de limitar el valor que pueda tener una propiedad autoimplementada. De esta manera, las propiedades autoimplementadas simplemente dejan que el nombre de las propiedades actúe como apoderado para el campo en sí. No obstante, esto es exactamente lo que se requiere en algunas ocasiones. De la misma manera, pueden ser de utilidad en los casos donde las propiedades se utilizan para exponer la funcionalidad de una tercera parte, quizás a través de una herramienta de diseño.

## Restricciones de las propiedades

Las propiedades tienen importantes restricciones. Primera, dado que una propiedad no define una locación de almacenamiento, no puede ser transmitida como parámetro **ref** ni **out** a un método. Segunda, no es posible sobrecargar una propiedad. (Sí puedes tener dos diferentes propiedades que accesen ambas a la misma variable subyacente, pero sería un caso inusual.) Finalmente, una propiedad no debe alterar el estado de la variable subyacente cuando se invoca el accesador **get**. Aunque esta regla no es puesta en vigor por el compilador, tal alteración es un error de sintaxis. Un operador **get** no debe crear efectos secundarios.

## Utilizar un modificador de acceso con un accesador

De forma predeterminada, **get** y **set** tienen la misma accesibilidad que los indexadores o propiedades de los cuales son parte. Por ejemplo, si la propiedad es declarada **public**, entonces, se establece que **get** y **set** sean públicos también. No obstante, es posible dar a **get** y **set** su propio modificador de acceso, como **private**. En todos los casos, el modificador de acceso para un accesador debe ser más restrictivo que la especificación de acceso de su propiedad o indexador.

Existen varias razones por las cuales querrás restringir la accesibilidad de un accesador. Por ejemplo, tal vez no quieras que nadie obtenga el valor de una propiedad, pero sí permitir que sólo los miembros de su propia clase establezcan la propiedad. Para hacer esto, declara el accesador **set** como **private**. Por ejemplo, he aquí una propiedad llamada **Max** que tiene un accesador **set** especificado como **private**:

```
class MiClase {
    int máximo;
    // ...
    public int Max {
        get {
            return máximo;
        }
        private set { // el accesador set es privado ← Observa que el accesador set
            if(value < 0) máximo = -value; es declarado privado.
            else máximo = value;
        }
    }
    // ...
}
```

Ahora, sólo el código dentro de **MiClase** puede establecer el valor de **Max**, pero cualquier código puede obtener su valor.

Tal vez el uso más importante de restringir el acceso a un accesador se encuentra cuando se trabaja con propiedades autoimplementadas. Como se explicó, no es posible crear una propiedad autoimplementada sólo-lectura o sólo-escritura, porque deben especificarse los dos accesadores, **get** y **set**, cuando es declarada. Sin embargo, puedes obtener el mismo efecto declarando cualquiera de las dos como **private**. El siguiente ejemplo declara lo que es, de hecho, una propiedad **Length** autoimplementada de sólo-lectura para la clase **ArregloFallaLeve** mostrado anteriormente:

```
public int Length { get; private set; }
```

Como **set** es **private**, **Length** puede ser establecida únicamente por el código dentro de su clase. Fuera de esa clase, cualquier intento por modificar **Length** es ilegal. Así, fuera de su clase, **Length** es de hecho sólo-lectura.

Para probar la versión autoimplementada de **Length** con **ArregloFallaLeve**, primero elimina la variable **len**. Luego, reemplaza cada uso de **len** dentro de **ArregloFallaLeve** con **Length**. He aquí una versión actualizada de esta clase, junto con **Main()** para demostrarlo:

```
// Muestra el funcionamiento de la propiedad Length autoimplementada en
// ArregloFallaLeve.

using System;

class ArregloFallaLeve {
    int[] a; // referencia al arreglo

    public bool ErrFlag; // indica el resultado de la última operación

    // Una propiedad Length autoimplementada sólo-lectura.
    public int Length { get; private set; } ← Una propiedad autoimplementada.
```

```
// Construye un arreglo de acuerdo con su tamaño.
public ArregloFallaLeve(int tamaño) {
    a = new int[tamaño];
    Length = tamaño; ← Asignación correcta a Length
    }
}

// Éste es el indexador para ArregloFallaLeve.
public int this[int index] {
    // Éste es el accesador get.
    get {
        if(ok(index)) {
            ErrFlag = false;
            return a[index];
        } else {
            ErrFlag = true;
            return 0;
        }
    }

    // Éste es el accesador set.
    set {
        if (ok(index)) {
            a[index] = value;
            ErrFlag = false;
        }
        else ErrFlag = true;
    }
}

// Regresa true si el índice está dentro de los límites.
private bool ok(int index) {
    if(index >= 0 & index < Length) return true;
    return false;
}

// Muestra la propiedad Length autoimplementada.
class AutoImpPropFSDemo {
    static void Main() {
        ArregloFallaLeve fs = new ArregloFallaLeve(5);
        int x;

        // Puede leer Length.
        for(int i=0; i < (fs.Length); i++)
            fs[i] = i*10;

        for(int i=0; i < (fs.Length); i++) {
            x = fs[i];
        }
    }
}
```

```

        if(x != -1) Console.WriteLine(x + " ");
    }
Console.WriteLine();

} // fs.Length = 10; // ¡Error! El accesador de Length es privado. ←
}                                     La asignación de Length fuera de ArregloFallaLeve es ilegal.
}

```

Esta versión de **ArregloFallaLeve** es igual a la de la versión anterior, pero no contiene un campo trasero explícitamente declarado.

## Prueba esto

## Crear una clase conjunto

Como se mencionó al principio de este capítulo, la sobrecarga de operadores, indexadores y propiedades ayuda a crear clases que pueden ser completamente integradas dentro del ambiente de programación de C#. Considera este punto: al definir los operadores, indexadores y propiedades necesarios, permities que un tipo de clase sea utilizado dentro de un programa de la misma manera como utilizarías un tipo integrado. Puedes actuar sobre objetos de esa clase a través de operadores e indexadores, y utilizar objetos de esa clase en expresiones. Añadir propiedades permite que la clase proporcione una interfaz consistente con los objetos integrados de C#. Para ilustrar la creación e integración de una nueva clase en el ambiente C#, crearemos una clase llamada **Conjunto** que defina, precisamente, un tipo conjunto.

Antes de comenzar, es importante comprender con precisión lo que entendemos por “conjunto”. Para los propósitos de este ejemplo, entenderemos, *conjunto* será una colección de elementos únicos. Es decir, dos elementos en un conjunto dado no pueden ser iguales. El orden de los miembros del conjunto es irrelevante. De tal manera que el conjunto

{A, B, C}

es el mismo que

{A, C, B}

Un conjunto también puede estar vacío.

El conjunto soporta cierta cantidad de operaciones. Las que implementaremos aquí serán:

- Añadir un elemento al conjunto.
- Eliminar un elemento del conjunto.
- Unión de conjuntos.
- Diferencia de conjuntos.

Añadir y eliminar elementos del conjunto son acciones que se explican por sí mismas. Las dos restantes merecen una explicación.

La *unión* de dos conjuntos da como resultado un tercero que contiene todos los elementos de ambos. (Por supuesto, no están permitidas las repeticiones.) Utilizaremos el operador + para realizar la unión de conjuntos.

(continúa)

La *diferencia* entre dos conjuntos da como resultado un tercero que contiene los elementos del primero que no son parte del segundo. Utilizaremos el operador – para realizar la diferencia de conjuntos. Por ejemplo, dados dos conjuntos S1 y S2, esta declaración elimina los elementos de S2 en S1, colocando el resultado en S3:

$$S3 = S1 - S2$$

Si S1 y S2 son el mismo, entonces S3 será un conjunto nulo.

Por supuesto, existen muchas más operaciones que se pueden realizar con conjuntos. Algunas son desarrolladas en la sección de Autoexamen. Otras te pueden servir de diversión al tratar de añadirlas por cuenta propia.

Por razones de sencillez, la clase **Conjunto** almacenará colecciones de caracteres, pero los mismos principios básicos pueden utilizarse para crear una clase **Conjunto** capaz de almacenar otro tipo de elementos.

## Paso a paso

1. Crea un nuevo archivo llamado **ConjuntoDemo.cs**.
2. Comienza creando **Conjunto** con las siguientes líneas:

```
class Conjunto {  
    char[] miembros; // este arreglo contiene el conjunto  
  
    // Una propiedad Length autoimplementada sólo-lectura.  
    public int Length { get; private set; }
```

Cada conjunto es almacenado en un arreglo **char** referido por **miembros**. La cantidad de miembros en el conjunto está representada por la propiedad autoimplementada **Length** (longitud). Observa que fuera de la clase **Conjunto**, es efectivamente sólo-lectura.

3. Añade los siguientes constructores de **Conjunto**:

```
// Construye un conjunto nulo.  
public Conjunto() {  
    Length = 0;  
}  
  
// Construye un conjunto vacío de un tamaño dado.  
public Conjunto(int tamaño) {  
    miembros = new char[tamaño]; // reserva memoria para el conjunto  
    Length = 0; // no tiene miembros cuando se construye  
}  
  
// Construye un conjunto a partir de otro.  
public Conjunto(Conjunto s) {  
    miembros = new char[s.Length]; // reserva memoria para el conjunto.  
    for(int i=0; i < s.Length; i++) miembros[i] = s[i];  
    Length = s.Length; // número de miembros  
}
```

Los conjuntos pueden ser construidos de tres maneras: Primera, se puede crear un conjunto nulo. Éste no contiene miembros, tampoco asigna ningún arreglo para miembros. Así, un conjunto nulo es simplemente un marcador de posición. Segunda, se puede crear un conjunto vacío de un tamaño dado. Finalmente, un conjunto puede ser construido a partir de otro. En este caso, los dos conjuntos contienen los mismos miembros, pero son referidos por objetos separados.

- 4.** Añade un indexador sólo-lectura, como se muestra aquí:

```
// Implementa un indexador sólo-lectura.
public char this[int idx] {
    get {
        if(idx >= 0 & idx < Length) return miembros[idx];
        else return (char)0;
    }
}
```

El indexador regresa un miembro de un conjunto dado su índice. Se realiza una verificación de límites para prevenir que el arreglo se desborde. Si el índice es inválido, se regresa un carácter nulo.

- 5.** Añade el método **encuentra( )** que se muestra a continuación. Este método determina si el elemento transmitido en **ch** es un miembro del conjunto. Regresa el índice del elemento si éste es localizado y -1 en caso de que el elemento no sea parte del conjunto. Observa que este miembro es privado.

```
/* Averigua si un elemento existe en un conjunto.
   Regresa el índice del elemento o -1 si no lo encuentra. */
int encuentra(char ch) {
    int i;

    for(i=0; i < Length; i++)
        if(miembros[i] == ch) return i;

    return -1;
}
```

- 6.** Comienza a añadir los operadores de conjunto, empezando con la adición. Para hacerlo, sobrecarga + para objetos de tipo **Conjunto**, como se muestra a continuación. Esta versión añade un elemento al conjunto.

```
// Añade un elemento único al conjunto.
public static Conjunto operator +(Conjunto ob, char ch) {

    // Si ch ya existe en el conjunto, regresa una copia
    // del conjunto original.
    if (ob.encuentra(ch) != -1) {

        // Regresa una copia del conjunto original.
        return new Conjunto(ob);

    } else { // Regresa un nuevo conjunto que contiene el nuevo elemento.
```

(continúa)

```
// Hace el nuevo conjunto un elemento más grande que el original.  
Conjunto nuevoconjunto = new Conjunto(ob.Length+1);  
  
// Copia elementos a un nuevo conjunto.  
for(int i=0; i < ob.Length; i++)  
    nuevoconjunto.miembros[i] = ob.miembros[i];  
  
// Establece la propiedad Length.  
nuevoconjunto.Length = ob.Length+1;  
  
// Añade un nuevo elemento a un conjunto nuevo.  
nuevoconjunto.miembros[nuevoconjunto.Length-1] = ch;  
  
return nuevoconjunto; // regresa el nuevo conjunto.  
}  
}
```

Este método merece un examen más cercano. Primero, el método **encuentra()** es invocado para determinar si **ch** ya es parte del conjunto. Si regresa cualquier otro valor diferente a **-1**, **ch** ya es parte del conjunto, por lo que regresa una copia del conjunto original. En caso contrario, se crea un conjunto nuevo, llamado **nuevoconjunto**, que guarda el contenido del conjunto original referido por **ob**, más el nuevo elemento **ch**. Observa que es creado un elemento más grande en comparación con **ob** para acomodar el nuevo elemento. A continuación, los elementos originales son copiados a **nuevoconjunto** y la longitud de éste se establece con una unidad mayor en comparación con el conjunto original. Paso siguiente, **ch** es añadido al final de **nuevoconjunto**. Finalmente, se regresa **nuevoconjunto**. En todos los casos, el conjunto original no es afectado por esta operación y un nuevo conjunto es regresado. De esta manera, el conjunto que regresa está separado y es distinto del conjunto transmitido como operando.

7. A continuación, sobrecarga – para que elimine un elemento de un conjunto, como se muestra aquí:

```
// Elimina un elemento del conjunto.  
public static Conjunto operator -(Conjunto ob, char ch) {  
    Conjunto nuevoconjunto = new Conjunto();  
    int i = ob.encuentra(ch); // i será -1 si el elemento no es  
    localizado  
  
    // Copia y comprime los elementos restantes.  
    for(int j=0; j < ob.Length; j++)  
        if(j != i) nuevoconjunto = nuevoconjunto + ob.miembros[j];  
  
    return nuevoconjunto;  
}
```

Primero se crea un nuevo conjunto nulo. Segundo, se invoca **encuentra()** para determinar si **ch** es miembro del conjunto original. Recuerda que **encuentra()** regresa **-1** si **ch** no es miembro. A continuación, los elementos del conjunto original se añaden al nuevo conjunto, excepto el elemento cuyo índice coincide con el regresado por **encuentra()**. Así, el conjunto resultante contiene todos los elementos del conjunto original, excepto por **ch**. Si **ch** no era parte del conjunto original con el que se comenzó, entonces los dos conjuntos son equivalentes.

- 8.** Sobrecarga el + y de nuevo el -, como se muestra aquí. Estas versiones ponen en funcionamiento la unión y diferenciación de conjuntos.

```
// Unión de conjuntos.  
public static Conjunto operator +(Conjunto ob1, Conjunto ob2) {  
    Conjunto nuevoconjunto = new Conjunto(ob1); // copia el primer  
    conjunto  
  
    // Añade elementos únicos del segundo conjunto.  
    for(int i=0; i < ob2.Length; i++)  
        nuevoconjunto = nuevoconjunto + ob2[i];  
  
    return nuevoconjunto; // regresa el conjunto actualizado  
}  
  
// Diferencia de conjuntos.  
public static Conjunto operator -(Conjunto ob1, Conjunto ob2) {  
    Conjunto nuevoconjunto = new Conjunto(ob1); // copia el primer  
    conjunto  
  
    // Resta elementos del segundo conjunto  
    for(int i=0; i < ob2.Length; i++)  
        nuevoconjunto = nuevoconjunto - ob2[i];  
  
    return nuevoconjunto; // regresa el conjunto actualizado.  
}
```

Como puedes ver, estos métodos utilizan las versiones de los operadores + y - definidas anteriormente para realizar sus operaciones. En el caso de la unión de conjuntos, se crea un nuevo conjunto que contiene los elementos del primero. Luego, los elementos del segundo conjunto son añadidos. Como la operación + sólo añade un elemento si éste no se encuentra ya en el conjunto, el conjunto resultante es la unión (sin duplicaciones) de los dos conjuntos. El operador de diferencia resta los elementos coincidentes.

- 9.** He aquí el código completo de la clase **Conjunto**, junto con la clase **ConjuntoDemo** que muestra su funcionamiento:

```
// Una clase conjunto para caracteres.  
  
using System;  
  
class Conjunto {  
    char[] miembros; // este arreglo contiene el conjunto  
  
    // Una propiedad Length autoimplementada sólo-lectura.  
    public int Length { get; private set; }  
  
    // Construye un conjunto nulo.  
    public Conjunto() {  
        Length = 0;  
    }
```

(continúa)

```
// Construye un conjunto vacío de un tamaño dado.
public Conjunto(int tamaño) {
    miembros = new char[tamaño]; // reserva memoria para el conjunto
    Length = 0; // no tiene miembros cuando se construye
}

// Construye un conjunto a partir de otro.
public Conjunto(Conjunto s) {
    miembros = new char[s.Length]; // reserva memoria para el conjunto
    for(int i=0; i < s.Length; i++) miembros[i] = s[i];
    Length = s.Length; // número de miembros
}

// Implementa un indexador sólo-lectura.
public char this[int idx] {
    get {
        if(idx >= 0 & idx < Length) return miembros[idx];
        else return (char)0;
    }
}

/* Averigua si un elemento existe en un conjunto.
   Regresa el índice del elemento o -1 si no lo encuentra. */
int encuentra(char ch) {
    int i;

    for(i=0; i < Length; i++)
        if(miembros[i] == ch) return i;

    return -1;
}

// Añade un elemento único al conjunto.
public static Conjunto operator +(Conjunto ob, char ch) {

    // Si ch ya existe en el conjunto, regresa una copia
    // del conjunto original.
    if(ob.encontrar(ch) != -1) {

        // Regresa una copia del conjunto original.
        return new Conjunto(ob);

    } else { // Regresa un nuevo conjunto que contiene el nuevo elemento.

        // Hace el nuevo conjunto un elemento más grande que el original.
        Conjunto nuevoconjunto = new Conjunto(ob.Length+1);
```

```
// Copia elementos a un nuevo conjunto.
for(int i=0; i < ob.Length; i++)
    nuevoconjunto.miembros[i] = ob.miembros[i];

// Establece la propiedad Length.
nuevoconjunto.Length = ob.Length+1;

// Añade un nuevo elemento a un conjunto nuevo.
nuevoconjunto.miembros[nuevoconjunto.Length-1] = ch;

return nuevoconjunto; // regresa el nuevo conjunto
}

// Elimina un elemento del conjunto.
public static Conjunto operator-(Conjunto ob, char ch) {
    Conjunto nuevoconjunto = new Conjunto();
    int i = ob.encuentra(ch); // i será -1 si el elemento no es
    localizado

    // Copia y comprime los elementos restantes.
    for(int j=0; j < ob.Length; j++)
        if(j != i) nuevoconjunto = nuevoconjunto + ob.miembros[j];

    return nuevoconjunto;
}

// Unión de conjuntos.
public static Conjunto operator +(Conjunto ob1, Conjunto ob2) {
    Conjunto nuevoconjunto = new Conjunto(ob1); // copia el primer
    conjunto

    // Añade elementos únicos del segundo conjunto.
    for(int i=0; i < ob2.Length; i++)
        nuevoconjunto = nuevoconjunto + ob2[i];

    return nuevoconjunto; // regresa el conjunto actualizado
}

// Diferencia de conjuntos.
public static Conjunto operator -(Conjunto ob1, Conjunto ob2) {
    Conjunto nuevoconjunto = new Conjunto(ob1); // copia el primer
    conjunto

    // Resta elementos del segundo conjunto
    for(int i=0; i < ob2.Length; i++)
        nuevoconjunto = nuevoconjunto - ob2[i];

    return nuevoconjunto; // regresa el conjunto actualizado.
}

}
```

(continúa)

```
// Muestra la clase Conjunto.
class ConjuntoDemo {
    static void Main() {
        // Construye un conjunto vacío de 10 elementos.
        Conjunto s1 = new Conjunto();
        Conjunto s2 = new Conjunto();
        Conjunto s3 = new Conjunto();

        s1 = s1 + 'A';
        s1 = s1 + 'B';
        s1 = s1 + 'C';

        Console.Write("s1 después de añadir A B C: ");
        for(int i=0; i<s1.Length; i++)
            Console.Write(s1[i] + " ");
        Console.WriteLine();

        s1 = s1 - 'B';
        Console.Write("s1 después de s1 = s1 - 'B': ");
        for(int i=0; i<s1.Length; i++)
            Console.Write(s1[i] + " ");
        Console.WriteLine();

        s1 = s1 - 'A';
        Console.Write("s1 después de s1 = s1 - 'A': ");
        for(int i=0; i<s1.Length; i++)
            Console.Write(s1[i] + " ");
        Console.WriteLine();

        s1 = s1 - 'C';
        Console.Write("s1 después de al = s1 - 'C': ");
        for(int i=0; i<s1.Length; i++)
            Console.Write(s1[i] + " ");
        Console.WriteLine("\n");

        s1 = s1 + 'A';
        s1 = s1 + 'B';
        s1 = s1 + 'C';
        Console.Write("s1 después de añadir A B C: ");
        for(int i=0; i<s1.Length; i++)
            Console.Write(s1[i] + " ");
        Console.WriteLine();

        s2 = s2 + 'A';
        s2 = s2 + 'X';
        s2 = s2 + 'W';
```

```
Console.WriteLine("s2 después de añadir A X W: ");
for(int i=0; i<s2.Length; i++)
    Console.Write(s2[i] + " ");
Console.WriteLine();

s3 = s1 + s2;
Console.WriteLine("s3 después de s3 = s1 + s2: ");
for(int i=0; i<s3.Length; i++)
    Console.Write(s3[i] + " ");
Console.WriteLine();

s3 = s3 - s1;
Console.WriteLine("s3 después de s3 - s1: ");
for(int i=0; i<s3.Length; i++)
    Console.Write(s3[i] + " ");
Console.WriteLine("\n");

s2 = s2 - s2; // limpia s2
s2 = s2 + 'C'; // añade ABC en orden inverso
s2 = s2 + 'B';
s2 = s2 + 'A';

Console.WriteLine("s1 es ahora: ");
for(int i=0; i<s1.Length; i++)
    Console.Write(s1[i] + " ");
Console.WriteLine();

Console.WriteLine("s2 es ahora: ");
for(int i=0; i<s2.Length; i++)
    Console.Write(s2[i] + " ");
Console.WriteLine();

Console.WriteLine("s3 es ahora: ");
for(int i=0; i<s3.Length; i++)
    Console.Write(s3[i] + " ");
Console.WriteLine();
}

}
```

Los datos de salida generados por el programa son:

```
s1 después de añadir A B C: A B C
s1 después de s1 = s1 - 'B': A C
s1 después de s1 = s1 - 'A': C
s1 después de a1 = s1 - 'C':
```

```
s1 después de añadir A B C: A B C
s2 después de añadir A X W: A X W
```

(continúa)

```
s3 después de s3 = s1 + s2: A B C X W  
s3 después de s3 - s1: X W
```

```
s1 es ahora: A B C  
s2 es ahora: C B A  
s3 es ahora: X W
```



### Autoexamen Capítulo 7

1. Muestra el formato general utilizado para sobrecargar un operador unitario. ¿De qué tipo debe ser el parámetro con relación a un método operador?
2. ¿Qué debes hacer para permitir operaciones que involucran un tipo de clase y un tipo integrado?
3. ¿Puede sobrecargarse el operador ?? ¿Puedes cambiar la precedencia de un operador?
4. ¿Qué es un indexador? Muestra su formato general.
5. ¿Qué funciones realizan los accesadores **get** y **set** dentro de un indexador?
6. ¿Qué es una propiedad? Muestra su formato general.
7. ¿Una propiedad define una ubicación de almacenamiento? En caso negativo, ¿dónde almacena un valor?
8. ¿Una propiedad puede ser transmitida como un argumento **ref** o **out**?
9. ¿Qué es una propiedad autoimplementada?
10. Define < y > para la clase **Conjunto** desarrollada en la sección *Prueba esto*, de tal manera que determinen si un conjunto es un subconjunto o un superconjunto de otro. Haz que < regrese **true** si el de la izquierda es un subconjunto del conjunto de la derecha y **false** de lo contrario. Haz que > regrese **true** si el de la izquierda es un superconjunto del conjunto de la derecha y **false** de lo contrario.
11. Define el operador **&** para la clase **Conjunto**, de tal manera que produzca la intersección de dos conjuntos.
12. Por cuenta propia, intenta añadir otros operadores **Conjunto**. Por ejemplo, intenta definir | de tal manera que produzca la *diferencia simétrica* entre dos conjuntos. (La diferencia simétrica consiste en aquellos elementos que los conjuntos no tienen en común.)

# Capítulo 8

## Herencia



## Habilidades y conceptos clave

- Fundamentos de la herencia
  - Acceso protegido
  - Invocar constructores de la clase base
  - Jerarquías de clase multinivel
  - Referencias de clases básicas para derivar objetos de clase
  - Métodos virtuales
  - Clases abstractas
  - **sealed**
  - La clase **object**
- 

**L**a herencia es uno de los tres principios fundamentales de la programación orientada a objetos porque permite la creación de clasificaciones jerárquicas. Utilizando la herencia, puedes crear una clase general que defina rasgos comunes a un conjunto de elementos relacionados. Luego, esta clase puede ser heredada a otras clases más específicas, añadiendo cada una de ellas los rasgos que las hacen únicas.

En el lenguaje C#, una clase que es heredada recibe el nombre de *clase base* y la clase que recibe la herencia se llama *clase derivada*. Por ello, una clase derivada es la versión especializada de una clase base. La derivada hereda todas las variables, métodos, propiedades e indexadores definidos por la base y añade sus propios y únicos elementos.

## Bases de la herencia

C# soporta la herencia al permitir que una clase incorpore otra clase dentro de su declaración. Esto se lleva a cabo especificando una clase base cuando es declarada una clase derivada. Comencemos con un ejemplo breve que ilustre varias características clave de la herencia. El siguiente ejemplo crea una clase base llamada **DosDForma** que almacena el ancho y alto de un objeto bidimensional, y crea una clase derivada llamada **Triángulo**. Pon especial atención a la manera en que se declara **Triángulo**.

```
// Una sencilla clase jerárquica.  
using System;  
  
// Una clase para objetos bidimensionales.  
class DosDForma {  
    public double Ancho;  
    public double Alto;
```

```
public void MuestraDim() {
    Console.WriteLine("Ancho y alto son " +
                      Ancho + " y " + Alto);
}

// Triángulo es derivada de DosDForma.
class Triángulo : DosDForma { ← Triángulo hereda DosDForma. Observa la sintaxis.
    public string Estilo;

    public double Área() { ← Triángulo puede hacer referencia a los miembros
        return Ancho * Alto / 2; ← de DosDForma como si fueran partes de Triángulo.
    }

    public void MuestraEstilo() {
        Console.WriteLine("Triángulo es " + Estilo);
    }
}

class Formas {
    static void Main() {
        Triángulo t1 = new Triángulo();
        Triángulo t2 = new Triángulo();

        t1.Ancho = 4.0; ← Todos los miembros de Triángulo están disponibles para los
        t1.Alto = 4.0; ← objetos Triángulo, incluso los heredados de DosDForma.
        t1.Esto = "isosceles";

        t2.Ancho = 8.0;
        t2.Alto = 12.0;
        t2.Esto = "recto";

        Console.WriteLine("Info para t1: ");
        t1.MuestraEstilo();
        t1.MuestraDim();
        Console.WriteLine("Área es " + t1.Área());

        Console.WriteLine();

        Console.WriteLine("Info para t2: ");
        t2.MuestraEstilo();
        t2.MuestraDim();
        Console.WriteLine("Área es " + t2.Área());
    }
}
```

Los datos generados por este programa son:

```
Info para t1:  
Triángulo es isósceles  
Ancho y alto son 4 y 4  
Área es 8
```

```
Info para t2:  
Triángulo es recto  
Ancho y alto son 8 y 12  
Área es 48
```

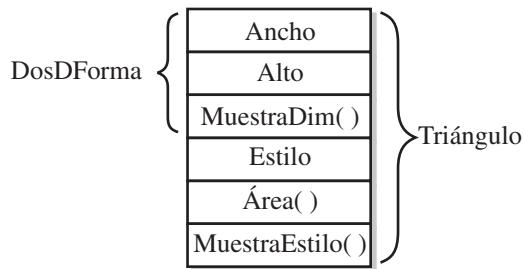
Aquí, **DosDForma** define los atributos genéricos de una figura de dos dimensiones, que puede ser cuadrado, rectángulo, triángulo y demás. La clase **Triángulo** crea un tipo específico de **DosDForma**, en este caso, un triángulo. La clase **Triángulo** incluye todas las características de **DosDForma** y añade el campo **Estilo**, el método **Área()** y el método **MuestraEstilo()**. **Estilo** almacena una descripción del tipo de triángulo. **Área()** calcula y regresa el área del triángulo y **MuestraEstilo()** muestra el estilo del triángulo.

Observa la sintaxis que se utiliza para heredar una clase base. El nombre de la clase base sigue al nombre de la clase derivada y están separadas por dos puntos (:). La sintaxis para heredar una clase es extremadamente sencilla y fácil de usar.

Dado que **Triángulo** incluye todos los miembros de su clase base, **DosDForma**, puede acceder a **Ancho** y **Alto** dentro de **Área()**. De la misma manera, dentro de **Main()**, los objetos **t1** y **t2** pueden hacer referencia directa a **Ancho** y **Alto**, como si fueran parte de **Triángulo**. La figura 8-1 representa de manera conceptual cómo se incorpora **DosDForma** a **Triángulo**.

Aunque **DosDForma** es la base de **Triángulo**, también es una clase completamente independiente y autónoma. Ser una clase base para una derivada no significa que la primera no pueda ser utilizada por sí misma. Por ejemplo, el siguiente código es perfectamente válido:

```
DosDForma forma = new DosDForma();  
  
forma.Ancho = 10;  
forma.Alto = 20;  
  
forma.MuestraDim();
```



**Figura 8-1** Una representación conceptual de la clase **Triángulo**

Por supuesto, un objeto **DosDForma** no tiene conocimiento de ninguna clase derivada de **DosDForma**, ni tampoco tiene acceso a ellas.

El formato general de una declaración de clase que hereda una clase base es el siguiente:

```
class clase-derivada : clase-base {
    // cuerpo de la clase
}
```

Observa que sólo puedes especificar una clase base para cualquier clase derivada que crees. C# no soporta la herencia de múltiples clases base en una sola clase derivada. (En esto difiere de C++, donde sí es posible heredar de múltiples clases base. Ten presente esta restricción cuando conviertas código C++ a C#.) No obstante, puedes crear una jerarquía de herencia en la cual una clase derivada se convierta en base de otra clase derivada. Por supuesto, ninguna clase puede ser base de sí misma. En todos los casos, una clase derivada hereda todos los miembros de su clase base. Éstos incluyen variables de instancia, métodos, propiedades e indexadores.

Una ventaja sobresaliente de la herencia es que una vez que has creado una clase que define los atributos comunes a un conjunto de objetos, se puede utilizar para crear cualquier cantidad de clases derivadas más específicas. Cada clase derivada puede ajustarse con más precisión a su propia clasificación. Por ejemplo, he aquí otra clase derivada de **DosDForma** que encapsula cuadriláteros:

```
// Una clase derivada de DosDForma para cuadriláteros.
class Cuadrilátero : DosDForma {
    // Regresa true si el cuadrilátero es un cuadrado.
    public bool EsCuadrado() {
        if(Ancho == Alto) return true;
        return false;
    }
    public double Area() {
        return Ancho * Alto;
    }
}
```

La clase **Cuadrilátero** incluye **DosDForma** y añade el método **EsCuadrado()**, el cual determina si el cuadrilátero es un cuadrado y **Área()**, que calcula el área del cuadrilátero.

## Acceso a miembros y herencia

Como aprendiste en el capítulo 6, los miembros de una clase por lo regular se declaran privados para prevenir su uso o intento de uso sin autorización. Heredar una clase *no* anula la restricción de acceso privado. De esta manera, aunque una clase derivada incluye todos los miembros de su clase base, no puede accesar aquellos miembros de la base que sean privados. Por ejemplo, si, como se muestra aquí, **Ancho** y **Alto** son declarados privados en **DosDForma**, entonces **Triángulo** no podrá tener acceso a ellos.

```
// El acceso a miembros privados no se hereda.
// Este ejemplo no se compilará.
using System;
```

```
// Una clase para objetos bidimensionales.
class DosDForma {
    double Ancho; // ahora privado
    double Alto; // ahora privado

    public void MuestraDim() {
        Console.WriteLine("Ancho y alto son " +
                           Ancho + " y " + Alto);
    }
}

// Triángulo es derivado de DosDForma.
class Triángulo : DosDForma {
    public string Estilo;
    No es posible acceder a miembros privados de la clase base.
    public double Area() {
        ↓
        return Ancho * Alto / 2; // ;Error! No es posible acceder a miembros
        privados.
    }

    public void MuestraEstilo() {
        Console.WriteLine("Triángulo es " + Estilo);
    }
}
```

La clase **Triángulo** no se compilará porque las referencias a **Ancho** y **Alto** dentro del método **Área()** provocan una violación de acceso. Dado que **Ancho** y **Alto** ahora son privados, sólo pueden ser accesados por otros miembros de su propia clase. Las clases derivadas no tienen acceso a ellos.

Recuerda: Un miembro privado de una clase permanecerá privativo de su propia clase. No es accesible por ningún código fuera de su clase, incluyendo las clases derivadas.

Al principio puedes pensar que es una restricción seria, el hecho de que las clases derivadas no puedan accesar a los miembros privados de su clase base, porque no permitiría el uso de los miembros privados en muchas situaciones. Pero no es verdad: C# proporciona varias soluciones. Una de ellas es utilizar miembros **protected** (protegidos), que se describen en la siguiente sección. Una segunda solución es utilizar propiedades públicas para proporcionar acceso a datos privados.

Como se explicó en el capítulo anterior, una propiedad te permite manejar el acceso a una variable de instancia. Por ejemplo, puedes aplicar restricciones sobre sus valores, o puedes hacer que la variable sea sólo-lectura. Al hacer una propiedad pública pero declarando su variable subyacente como privada, una clase derivada aún puede hacer uso de la propiedad, pero no puede accesar directamente la variable privada subyacente.

Aquí está una nueva versión de la clase **DosDForma** que convierte **Ancho** y **Alto** en propiedades. En el proceso, asegura que los valores de **Ancho** y **Alto** sean positivos. Esto te permitirá, por ejemplo, especificar el **Ancho** y **Alto** utilizando las coordenadas de la figura en cualquier cuadrante del plano cartesiano sin tener que obtener primero sus valores absolutos.

```
// Utiliza propiedades para establecer y recuperar miembros privados.
using System;
```

```
// Una clase para objetos bidimensionales.  
class DosDForma {  
    double pri_ancho;  
    double pri_alto;  
  
    // Propiedades para Ancho y Alto.  
    public double Ancho { ←  
        get { return pri_ancho; }  
        set { pri_ancho = value < 0 ? -value : value; } } ← Aquí, Ancho y Alto  
    } → son propiedades.  
  
    public double Alto { ←  
        get { return pri_alto; }  
        set { pri_alto = value < 0 ? -value : value; } } ←  
    } →  
  
    public void MuestraDim() {  
        Console.WriteLine("Ancho y alto son " +  
                           Ancho + " y " + Alto);  
    } ←  
}  
  
// Una clase derivada para triángulos DosDForma.  
class Triángulo : DosDForma {  
    public string Estilo;  
  
    public double Area() { ←  
        return Ancho * Alto / 2; ← Uso de Ancho y Alto es OK porque  
    } → el acceso se hace por propiedades.  
  
    public void MuestraEstilo() {  
        Console.WriteLine("Triángulo es " + Estilo);  
    } ←  
}  
  
class Formas2 {  
    static void Main() {  
        Triángulo t1 = new Triángulo();  
        Triángulo t2 = new Triángulo();  
  
        t1.Ancho = 4.0;  
        t1.Alto = 4.0;  
        t1.Estilo = "isósceles";  
  
        t2.Ancho = 8.0;  
        t2.Alto = 12.0;  
        t2.Estilo = "recto";
```

```

Console.WriteLine("Info para t1: ");
t1.MuestraEstilo();
t1.MuestraDim();
Console.WriteLine("Área es " + t1.Área());

Console.WriteLine();
Console.WriteLine("Info para t2: ");
t2.MuestraEstilo();
t2.MuestraDim();
Console.WriteLine("Área es " + t2.Área());
}
}

```

Este programa muestra los mismos datos de salida que la versión anterior. La única diferencia es que ahora **Ancho** y **Alto** son propiedades públicas que manejan acceso a variables de instancia privadas.

## Pregunta al experto

**P:** He escuchado los términos “superclase” y “subclase” utilizados en discusiones sobre la programación en Java. ¿Estos términos tienen algún significado en C#?

**R:** Lo que en Java se llama superclase, en C# recibe el nombre de clase base. Lo que en Java se conoce como subclase, en C# recibe el nombre de clase derivada. Por lo regular escucharás ambos juegos de términos aplicados a las clases de ambos programas, pero este libro continuará utilizando los términos estándar de C#. Por cierto, C++ también utiliza la terminología clase base y clase derivada.

## Utilizar acceso protegido

Como se explicó en la sección anterior, un miembro privado de una clase base no es accesible por una clase derivada. Esto parecería implicar que si quisieras que una clase derivada tuviera acceso a algún miembro en la base, necesitaría ser público. Por supuesto, al declararlo público también estaría disponible para cualquier otro código, lo cual podría no ser deseable. Por fortuna, esta implicación está equivocada porque C# te permite la creación de *miembros protegidos*. Un miembro protegido es accesible dentro de su jerarquía de clase, pero privado fuera de esa jerarquía.

Un miembro protegido se crea utilizando el modificador de acceso **protected**. Cuando un miembro de la clase es declarado **protected**, este miembro es privado, con una importante excepción. La excepción ocurre cuando un miembro protegido es heredado. En este caso, el miembro protegido de la clase base se convierte en un miembro protegido de la clase derivada y, por tanto, accesible para esta última. De esta manera, utilizando **protected** puedes crear miembros de clase que sean privativos de la clase, pero que puedan ser heredados y accesados por una clase derivada.

He aquí un ejemplo que utiliza **protected**:

```
// Muestra el uso de protected.
using System;
class B {
    protected int i, j; // privativo de B, pero accesible para D
    public void Configura(int a, int b) {
        i = a;
        j = b;
    }
    public void Muestra() {
        Console.WriteLine(i + " " + j);
    }
}
class D : B {
    int k; // privado

    // D puede accesar i y j de B.
    public void ConfiguraK() {
        k = i * j; ← D puede accesar i y j porque son protegidos, no privados.
    }
    public void Muestrak() {
        Console.WriteLine(k);
    }
}

class ProtectedDemo {
    static void Main() {
        D ob = new D();

        ob.Configura(2, 3); // OK, conocido por D
        ob.Muestra();      // OK, conocido por D

        ob.ConfiguraK(); // OK, parte de D
        ob.Muestrak(); // OK, parte de D
    }
}
```

En este ejemplo, dado que **B** es heredada por **D** y dado que **i** y **j** son declaradas como **protected** en **B**, el método **ConfiguraK()** puede accesarlas. Si **i** y **j** fueran declaradas como privadas por **B**, entonces **D** no tendría acceso a ellas y el programa no se compilaría.

Como **public** y **private**, el estatus **protected** permanece con el miembro sin importar cuántos estratos de herencia estén involucrados. Por ello, cuando una clase derivada es utilizada como base

para otra derivada, todo miembro **protected** de la clase base inicial que es heredado por la primera derivada, también es heredado como **protected** por la segunda derivada.

Aunque el acceso **protected** es muy útil, no se aplica en todas las situaciones. Por ejemplo, en el caso de **DosDForma** mostrado en la sección anterior, queríamos explícitamente que **Ancho** y **Alto** fueran valores de acceso público, porque queríamos manipular los valores que les son asignados. Por lo mismo, declararlas **protected** no es una opción válida. En este caso, el uso de propiedades proporciona una solución apropiada, al controlar en lugar de prevenir el acceso. Recuerda, utiliza **protected** cuando quieras crear un miembro que sea privado a lo largo de una jerarquía de clase, pero que por lo demás no tiene restricciones. Para manejar el acceso de un valor, utiliza propiedades.

## Constructores y herencia

En una jerarquía, tanto las clases base como las derivadas pueden tener sus propios constructores. Esto hace surgir una importante pregunta: ¿Qué constructor es el responsable de construir un objeto de la clase derivada? ¿El constructor en la clase base, el de la clase derivada, o ambos? La respuesta es la siguiente: El constructor de la base construye la porción correspondiente a la misma y el constructor de la derivada se encarga de construir su porción correspondiente. Esto tiene sentido porque la clase base no tiene conocimiento de ningún elemento de la derivada, ni acceso a ellos. Por ello, sus constructores deben actuar por separado. Los ejemplos anteriores se han apoyado sobre los constructores predeterminados creados en automático por C#, por lo que no hubo ningún problema. Sin embargo, en la práctica, la mayoría de las clases tendrán constructores. A continuación veremos cómo manejar esta situación.

Cuando sólo la clase derivada define un constructor, el proceso es muy directo: simplemente se construye el objeto de la clase derivada. La porción de la clase base se construye automáticamente utilizando su constructor predeterminado. Por ejemplo, aquí tenemos una versión modificada de **Triángulo** que define un constructor. También declara **Estilo** como privada porque ahora es establecida por el constructor.

```
// Añade un constructor a Triángulo.
using System;

// Una clase para objetos bidimensionales.
class DosDForma {
    double pri_ancho;
    double pri_alto;

    // Propiedades para Ancho y Alto.
    public double Ancho {
        get { return pri_ancho; }
        set { pri_ancho = value < 0 ? -value : value; }
    }

    public double Alto {
        get { return pri_alto; }
        set { pri_alto = value < 0 ? -value : value; }
    }
}
```

```
public void MuestraDim() {
    Console.WriteLine("Ancho y alto son " +
                      Ancho + " y " + Alto);
}

// Una clase derivada de DosDForma para triángulos.
class Triángulo : DosDForma {
    string Estilo;

    // Constructor
    public Triángulo(string s, double w, double h) { ← Constructor para Triángulo.
        Ancho = w; // inicia la clase base ← Construye la porción DosDForma
        Alto = h; // inicia la clase base ← para el objeto Triángulo.

        Estilo = s; // inicia la clase derivada
    }

    public double Área() {
        return Ancho * Alto / 2;
    }

    public void MuestraEstilo() {
        Console.WriteLine("Triángulo es " + Estilo);
    }
}

class Formas3 {
    static void Main() {
        Triángulo t1 = new Triángulo("isósceles", 4.0, 4.0);
        Triángulo t2 = new Triángulo("recto", 8.0, 12.0);

        Console.WriteLine("Info para t1: ");
        t1.MuestraEstilo();
        t1.MuestraDim();
        Console.WriteLine("Área es " + t1.Área());

        Console.WriteLine();

        Console.WriteLine("Info para t2: ");
        t2.MuestraEstilo();
        t2.MuestraDim();
        Console.WriteLine("Área es " + t2.Área());
    }
}
```

Aquí, el constructor de **Triángulo** inicializa los miembros de **DosDForma** que hereda, junto con su propio campo **Estilo**. Los datos de salida son los mismos que en el ejemplo anterior.

Cuando ambas clases, la base y la derivada, definen constructores, el proceso es un poco más complejo porque ambos constructores se deben ejecutar. En este caso debes utilizar otra palabra clave de C#: **base**, que tiene dos usos. El primero invoca un constructor de la clase base. El segundo es para accesar un miembro de la clase base que ha sido ocultado por un miembro de una clase derivada. Aquí veremos su primer uso.

## Invocar constructores de la clase base

Una clase derivada puede invocar un constructor definido en su clase base utilizando un formato expandido de su declaración para el constructor de clase derivada y la palabra clave **base**. El formato general de esta declaración expandida se muestra a continuación:

```
constructor-derivado(lista-parámetros) : base(lista-args) {  
    // cuerpo del constructor  
}
```

Aquí, *lista-args* especifica cualquier argumento que necesite el constructor en la clase base. Observa la colocación de los dos puntos.

Para ver cómo se utiliza **base**, analiza la versión de **DosDForma** en el siguiente programa. Define un constructor que inicializa las propiedades **Ancho** y **Alto**. Luego, el constructor es invocado por el constructor de **Triángulo**.

```
// Añade constructores a DosDFormas.  
using System;  
  
// Una clase para objetos bidimensionales.  
class DosDForma {  
    double pri_ancho;  
    double pri_alto;  
  
    // Constructor para DosDForma.  
    public DosDForma(double w, double h) { ←———— Constructor para DosDFormas.  
        Ancho = w;  
        Alto = h;  
    }  
  
    // Propiedades para Ancho y Alto.  
    public double Ancho {  
        get { return pri_ancho; }  
        set { pri_ancho = value < 0 ? -value : value; }  
    }  
  
    public double Alto {  
        get { return pri_alto; }  
        set { pri_alto = value < 0 ? -value : value; }  
    }  
}
```

```

public void MuestraDim() {
    Console.WriteLine("Ancho y alto son " +
                      Ancho + " y " + Alto);
}
}

// Una clase derivada de DosDForma para triángulos.
class Triángulo : DosDForma {
    string Estilo;

    // Invoca el constructor de la clase base.
    public Triángulo(string s, double w, double h) : base(w, h) {
        Estilo = s;
    }
}

public double Área() {
    return Ancho * Alto / 2;
}

public void MuestraEstilo() {
    Console.WriteLine("Triángulo es " + Estilo);
}
}

class Formas4 {
    static void Main() {
        Triángulo t1 = new Triángulo("isósceles", 4.0, 4.0);
        Triángulo t2 = new Triángulo("recto", 8.0, 12.0);

        Console.WriteLine("Info para t1: ");
        t1.MuestraEstilo();
        t1.MuestraDim();
        Console.WriteLine("Área es " + t1.Área());

        Console.WriteLine();

        Console.WriteLine("Info para t2: ");
        t2.MuestraEstilo();
        t2.MuestraDim();
        Console.WriteLine("Área es " + t2.Área());
    }
}

```

↑  
Utiliza **base** para ejecutar el constructor **DosDFormas**.

Aquí, **Triángulo( )** invoca **base** con los parámetros **w** y **h**. Esto provoca que el constructor **DosDForma( )** sea invocado, lo cual inicializa **Ancho** y **Alto** utilizando estos valores. **Triángulo** ya no inicializa estos valores por sí mismo. Sólo necesita inicializar el valor único en él: **Estilo**. Esto deja libre a **DosDForma** para que construya sus subobjetos de la manera que elija. Más aún, **DosDForma** puede añadir funcionalidad sobre la cual las clases derivadas existentes no tienen conocimiento, lo cual previene que se rompa el código ya existente.

Cualquier forma de constructor definido por la clase base puede ser invocado por la palabra clave **base**. El constructor que se ejecuta será el que coincida con los argumentos. Por ejemplo, a continuación presentamos versiones expandidas de **DosDForma** y **Triángulo** que incluyen constructores predeterminados y constructores que toman un argumento.

```
// Añade más constructores a DosDForma.
using System;

class DosDForma {
    double pri_ancho;
    double pri_alto;

    // Constructor predeterminado.
    public DosDForma() {
        Ancho = Alto = 0.0;
    }

    // Especifica Ancho y Alto.
    public DosDForma(double w, double h) {
        Ancho = w;
        Alto = h;
    }

    // Construye objeto con ancho y alto iguales.
    public DosDForma(double x) {
        Ancho = Alto = x;
    }

    // Propiedades para Ancho y Alto.
    public double Ancho {
        get { return pri_ancho; }
        set { pri_ancho = value < 0 ? -value : value; }
    }

    public double Alto {
        get { return pri_alto; }
        set { pri_alto = value < 0 ? -value : value; }
    }

    public void MuestraDim() {
        Console.WriteLine("Ancho y alto son " +
                          Ancho + " y " + Alto);
    }
}

// Una clase derivada de DosDForma para triángulos.
class Triángulo : DosDForma {
    string Estilo;
```

```
/* Un constructor predeterminado. Invoca automáticamente
   el constructor predeterminado de DosDForma. */
public Triángulo() {
    Estilo = "null";
}

// Constructor que toma estilo, ancho y alto.
public Triángulo(string s, double w, double h) : base(w, h) {
    Estilo = s;
}

// Construye un triángulo isósceles.
public Triángulo(double x) : base(x) { ←
    Estilo = "isósceles";
}

public double Área() {
    return Ancho * Alto / 2;
}

public void MuestraEstilo() {
    Console.WriteLine("Triángulo es " + Estilo);
}
}

class Formas5 {
static void Main() {
    Triángulo t1 = new Triángulo();
    Triángulo t2 = new Triángulo("recto", 8.0, 12.0);
    Triángulo t3 = new Triángulo(4.0);

    t1 = t2;

    Console.WriteLine("Info para t1: ");
    t1.MuestraEstilo();
    t1.MuestraDim();
    Console.WriteLine("Área es " + t1.Área());

    Console.WriteLine();

    Console.WriteLine("Info para t2: ");
    t2.MuestraEstilo();
    t2.MuestraDim();
    Console.WriteLine("Área es " + t2.Área());

    Console.WriteLine();

    Console.WriteLine("Info para t3: ");
    t3.MuestraEstilo();
```

Usa **base** para invocar diferentes formas del constructor **DosDForma**.

```
t3.MuestraDim();
Console.WriteLine("Área es " + t3.Área());
Console.WriteLine();
}
}
```

Aquí están los datos de salida generados por esta nueva versión:

```
Info para t1:
Triángulo es recto
Ancho y alto son 8 y 12
Área es 48

Info para t2:
Triángulo es recto
Ancho y alto son 8 y 12
Área es 48

Info para t3:
Triángulo es isósceles
Ancho y alto son 4 y 4
Área es 8
```

Revisemos los conceptos clave detrás de **base**. Cuando una clase derivada especifica una cláusula **base**, está invocando el constructor de su clase base inmediata. Así, **base** siempre hace referencia a la base inmediata superior de la clase que invoca. Esto se aplica incluso en una jerarquía de niveles múltiples. Los argumentos se transmiten al constructor base especificándolos como argumentos para **base**. Si no está presente ninguna cláusula **base**, entonces se invoca en automático el constructor predeterminado de la clase base.

## Herencia y ocultamiento de nombres

Las clases derivadas pueden definir un miembro que tenga el mismo nombre que un miembro de su clase base. Cuando esto sucede, el miembro de la base se oculta dentro de la clase derivada. Aunque esto no es técnicamente un error en C#, el compilador enviará un mensaje de advertencia. Tales alertas te indican que un nombre ha sido ocultado. Si tu intención es ocultar un miembro de la clase base, entonces, para prevenir estas advertencias, el miembro de la clase derivada debe ser precedido por la palabra clave **new**. Debe quedar claro que este uso de **new** es completamente diferente y sin relación con su uso para crear una instancia objeto.

He aquí un ejemplo de ocultamiento de nombre:

```
// Un ejemplo de ocultamiento de nombre relacionado con la herencia.
using System;

class A {
    public int i = 0;
}
```

```
// Crea una clase derivada.
class B : A {
    new int i; // esta i oculta la i en A ← Esta i en A es ocultada por la i
    en B. Observa el uso de new.

    public B(int b) {
        i = b; // i en B
    }

    public void Muestra() {
        Console.WriteLine("i en la clase derivada: " + i);
    }
}

class OcultaNombre {
    static void Main() {
        B ob = new B(2);

        ob.Muestra();
    }
}
```

Primero, observa el uso de **new**. En esencia, le indica al compilador que sabes que una nueva variable llamada **i** está siendo creada y que oculta la **i** en la clase base **A**. Si dejas fuera **new**, se genera una advertencia.

Los datos generados por este programa son los siguientes:

```
i en la clase derivada: 2
```

Como **B** define su propia variable de instancia llamada **i**, ésta oculta la **i** en **A**. Por tanto, cuando **Muestra()** es invocado sobre un objeto de tipo **B**, se muestra el valor de **i** definido por **B**, no el definido por **A**.

## Utilizar base para accesar un nombre oculto

Existe un segundo formato de **base** que de alguna manera actúa como **this**, excepto que siempre hace referencia a la clase base derivada en la cual se usa. Este uso tiene el siguiente formato general:

**base.miembro**

Aquí, *miembro* puede ser un método o una variable de instancia. Este formato de **base** es más aplicable a situaciones en las cuales los nombres de los miembros de una clase derivada ocultan otros miembros con el mismo nombre en la clase base. Analiza esta versión de la clase jerárquica del ejemplo anterior:

```
// Utiliza base para superar un nombre oculto.
using System;

class A {
    public int i = 0;
}
```

## 304 Fundamentos de C# 3.0

---

```
// Crea una clase derivada.  
class B : A {  
    new int i; // esta i oculta la i en A  
  
    public B(int a, int b) {  
        base.i = a; // esto descubre la i en A ← Aquí, base.i hace referencia a la i en A.  
        i = b; // i en B  
    }  
  
    public void Muestra() {  
        // Esto muestra la i en A.  
        Console.WriteLine("i en clase base: " + base.i);  
  
        // Esto muestra la i en B.  
        Console.WriteLine("i en clase derivada: " + i);  
    }  
}  
  
class DescubreNombre {  
    static void Main() {  
        B ob = new B(1, 2);  
  
        ob.Muestra();  
    }  
}
```

El programa genera los siguientes datos de salida:

```
i en clase base: 1  
i en clase derivada: 2
```

Aunque la variable de instancia **i** en **B** oculta **i** en **A**, **base** permite el acceso a la **i** definida en la clase base.

Los métodos ocultos también pueden ser invocados a través del uso de **base**. Por ejemplo:

```
// Invoca un método oculto.  
using System;  
  
class A {  
    public int i = 0;  
  
    // Muestra() en A.  
    public void Muestra() {  
        Console.WriteLine("i en la clase base: " + i);  
    }  
}  
  
// Crea una clase derivada.  
class B : A {  
    new int i; // esta i oculta la i en A
```

```

public B(int a, int b) {
    base.i = a; // esto descubre la i en A
    i = b; // i en B
}

// Esto oculta Muestra() en A.
new public void Muestra() { ← Este método Muestra() oculta el de A.
    base.Muestra(); // esto invoca Muestra() en A ← Esto invoca el método
                    Muestra() oculto.

    // Esto despliega la i en B.
    Console.WriteLine("i en la clase derivada: " + i);
}
}

class DescubreNombre {
    static void Main() {
        B ob = new B(1, 2);

        ob.Muestra();
    }
}

```

Los datos de salida generados por este programa son:

```
i en la clase base: 1
i en la clase derivada: 2
```

Como puedes ver, **base.Muestra()** invoca la versión de la clase base de **Muestra()**.

Otro punto: observa que **new** es utilizada en este programa para indicar al compilador que sabes que un nuevo método llamado **Muestra()** está siendo creado y que oculta al método **Muestra()** en **A**.

## Prueba esto

## Extender la clase Vehículo

Para mostrar el poder de la herencia, extenderemos la clase **Vehículo** desarrollada en el capítulo 4. Como debes recordar, **Vehículo** encapsula información sobre vehículos, incluyendo el número de pasajeros que pueden transportar, su capacidad en el tanque de gasolina y el promedio de consumo de combustible. Podemos utilizar la clase **Vehículo** como punto de partida para desarrollar clases más especializadas. Por ejemplo, un tipo de vehículo es un camión. Un atributo importante de los camiones es su capacidad de carga. Así, para crear una clase **Camión**, puedes heredar **Vehículo** y añadir una variable de instancia que almacene la capacidad de carga. En este proyecto crearás la clase **Camión**. En el proceso, las variables de instancia en **Vehículo** serán convertidas en propiedades autoimplementadas en las cuales los accesores **set** serán especificados como **protected**. Esto significa que sus valores pueden ser establecidos por código en la clase derivada, pero no son accesibles de ninguna otra manera.

(continúa)

## Paso a paso

1. Crea un archivo llamado **CamiónDemo.cs** y copia la última implementación de **Vehículo** del capítulo 4 a ese archivo.
2. Convierte las variables de instancia de **Vehículo** en propiedades autoimplementadas. Especifica el accesador **set** como **protected**, como se muestra a continuación:

```
/* Propiedades autoimplementadas para pasajeros,
   capacidad de gasolina y kilometraje. Observa que el
   accesador set está protegido. */
public int Pasajeros { get; protected set; }

public int Capacidad { get; protected set; }

public int Kpl { get; protected set; }
```

Como se mencionó, hacer los accesorios **set protected** significa que serán accesibles por cualquier clase derivada de **Vehículo**, pero por lo demás serán privados.

3. Crea la clase **Camión** como se muestra aquí:

```
// Usa Vehículo para crear una especialización Camión.
class Camión : Vehículo {

    // Éste es un constructor para Camión.
    public Camión(int p, int f, int m, int c) : base(p, f, m)
    {
        CargaCap = c;
    }

    // Propiedad autoimplementada para capacidad de carga en kilos.
    public int CargaCap { get; protected set; }
}
```

Aquí, **Camión** hereda **Vehículo** y añade la propiedad **CargaCap**. Así, **Camión** incluye todos los atributos generales definidos por **Vehículo**. Sólo necesita añadir aquellos elementos que son únicos de su propia clase. Observa que **CargaCap** es también una propiedad autoimplementada cuyos accesadores **set** se especifican como **protected**. Esto significa que será accesible a cualquier clase que herede **Camión**, pero en cualquier otro caso es privada.

4. He aquí el programa completo que muestra el funcionamiento de la clase **Camión**:

```
// Construye una clase derivada de Vehículo para Camiones.

using System;

class Vehículo {

    // Éste es un constructor de Vehículo.
    public Vehículo(int p, int f, int m) {
        Pasajeros = p;
```

```
Capacidad = f;
Kpl = m;
}

// Regresa promedio.
public int Rango() {
    return Kpl * Capacidad;
}

// Calcula la gasolina necesaria para una distancia dada.
public double GasNecesaria(int kms) {
    return (double)kms / Kpl;
}

/* Propiedades autoimplementadas para pasajeros,
   capacidad de gasolina y kilometraje. Observa que el
   accesador set está protegido. */
public int Pasajeros { get; protected set; }

public int Capacidad { get; protected set; }

public int Kpl { get; protected set; }
}

// Usa Vehículo para crear una especialización Camión.
class Camión : Vehículo {

    // Éste es un constructor para Camión.
    public Camión(int p, int f, int m, int c) : base(p, f, m)
    {
        CargaCap = c;
    }

    // Propiedad autoimplementada para capacidad de carga en kilos.
    public int CargaCap { get; protected set; }
}

class CamiónDemo {
    static void Main() {

        // Construye algunos camiones.
        Camión semi = new Camión(2, 200, 3, 44000);
        Camión pickup = new Camión(3, 28, 6, 2000);

        double litros;
        int dist = 252;

        litros = semi.GasNecesaria(dist);
```

(continúa)

```
Console.WriteLine("Semirremolque puede cargar " + semi.CargaCap +
                  " kilos.");
Console.WriteLine("Para recorrer " + dist + " kilómetros
                  semirremolque necesita " + litros + " litros
                  de combustible.\n");

litros = pickup.GasNecesaria(dist);

Console.WriteLine("Pickup puede cargar " + pickup.CargaCap +
                  " kilos.");
Console.WriteLine("Para recorrer " + dist + " kilómetros pickup
                  necesita " + litros + " litros de combustible.");
    }
}
```

5. Los datos de salida generados por este programa son:

```
Semirremolque puede cargar 44000 kilos.
Para recorrer 252 kilómetros semirremolque necesita 84 litros de
combustible.

Pickup puede cargar 2000 kilos.
Para recorrer 252 kilómetros pickup necesita 42 litros de combustible.
```

Muchos otros tipos de clases pueden derivar de **Vehículo**. Por ejemplo, el siguiente esquema crea una clase de vehículos limpiadores de carreteras que almacena el terreno limpiado por el vehículo:

```
// Crea una clase de vehículo limpiador.
class limpiador : Vehículo {
    // Terreno limpiado en metros.
    public int TerrenoLimpiado { get; protected set }
    // ...
}
```

El punto clave es: una vez que has creado una clase base que define los aspectos generales de un objeto, la base puede ser heredada a clases especializadas. Cada clase derivada simplemente añade sus propios atributos característicos y únicos. En esencia, eso es la herencia.

---

## Crear una jerarquía multinivel

Hasta este punto hemos utilizado jerarquías de clase sencillas, consistentes en una clase base y una derivada. Sin embargo, puedes construir jerarquías que contengan tantas capas de herencia como gustes. Como ya se mencionó, es perfectamente aceptable utilizar una clase derivada como base para otra. Por ejemplo, dadas tres clases **A**, **B** y **C**, **C** puede ser derivada de **B**, que a su vez puede ser derivada de **A**. Cuando ocurren este tipo de situaciones, cada clase derivada hereda todos los rasgos encontrados en todas sus clases base. En este caso, **C** heredaría todos los aspectos de **B** y **A**.

Para ver la utilidad de la jerarquía multinivel, examina el siguiente programa. En él, la clase derivada **Triángulo** es utilizada como base para crear la clase derivada llamada **ColorTriángulo**. Esta última hereda todos los rasgos de **Triángulo** y de **DosDForma**, y añade un campo llamado **Color**, que contiene el color del triángulo.

```
// Una jerarquía multinivel.
using System;

class DosDForma {
    double pri_ancho;
    double pri_alto;

    // Constructor predeterminado.
    public DosDForma() {
        Ancho = Alto = 0.0;
    }

    // Constructor que toma ancho y alto.
    public DosDForma(double w, double h) {
        Ancho = w;
        Alto = h;
    }

    // Especifica ancho y alto.
    public DosDForma(double x) {
        Ancho = Alto = x;
    }

    // Propiedades de Ancho y Alto.
    public double Ancho {
        get { return pri_ancho; }
        set { pri_ancho = value < 0 ? -value : value; }
    }

    public double Alto {
        get { return pri_alto; }
        set { pri_alto = value < 0 ? -value : value; }
    }

    public void MuestraDim() {
        Console.WriteLine("Ancho y alto son " +
                          Ancho + " y " + Alto);
    }
}

// Una clase derivada de DosDForma para triángulos.
class Triángulo : DosDForma {
    string Estilo;
```

## 310 Fundamentos de C# 3.0

---

```
/* Un constructor predeterminado. Invoca el constructor
predeterminado de DosDForma. */
public Triángulo() {
    Estilo = "null";
}

// Constructor que toma estilo, ancho y alto.
public Triángulo(string s, double w, double h) : base(w, h) {
    Estilo = s;
}

// Construye un triángulo isósceles.
public Triángulo(double x) : base(x) {
    Estilo = "isósceles";
}

public double Area() {
    return Ancho * Alto / 2;
}

public void ShowStyle() {
    Console.WriteLine("Triángulo es " + Estilo);
}

// Extiende Triángulo.
class ColorTriángulo : Triángulo { ←
    string Color;

    // Constructor para ColorTriángulo.
    public ColorTriángulo(string c, string s,
        double w, double h) : base(s, w, h) {
        Color = c;
    }

    // Muestra el color.
    public void MuestraColor() {
        Console.WriteLine("Color es " + Color);
    }
}

class Formas6 {
    static void Main() {
        ColorTriángulo t1 =
            new ColorTriángulo("Azul", "recto", 8.0, 12.0);
        ColorTriángulo t2 =
            new ColorTriángulo("Rojo", "isósceles", 2.0, 2.0);
    }
}
```

**ColorTriángulo** hereda **Triángulo**, que es descendiente de **DosDForma**. Por tanto, incluye todos los miembros de ambos.

```

Console.WriteLine("Info para t1: ");
t1.ShowStyle();
t1.MuestraDim(); ← Un objeto ColorTriángulo puede llamar métodos
t1.MuestraColor();
Console.WriteLine("Área es " + t1.Área());

Console.WriteLine();

Console.WriteLine("Info para t2: ");
t2.ShowStyle();
t2.MuestraDim();
t2.MuestraColor();
Console.WriteLine("Área es " + t2.Área());
}
}

```

Los datos de salida generados por el programa son:

```

Info para t1:
Triángulo es recto
Ancho y alto son 8 y 12
Color es Azul
Área es 48

```

```

Info para t2:
Triángulo es isósceles
Ancho y alto son 2 y 2
Color es Rojo
Área es 2

```

Debido a la herencia, **ColorTriángulo** puede hacer uso de las clases **Triángulo** y **DosDForma**, previamente definidas, añadiendo sólo la información extra que necesita para su propia aplicación específica. Esto es parte del valor de la herencia; te permite reutilizar código.

Este ejemplo ilustra otro punto importante: **base** siempre hace referencia al constructor de su base inmediata anterior. La palabra clave **base** en **ColorTriángulo** invoca al constructor en **Triángulo**. La palabra clave **base** en **Triángulo** invoca al constructor de **DosDForma**. En una jerarquía de clases, si el constructor de una clase base requiere parámetros, entonces todas las clases derivadas deben transigir esos parámetros “sobre la línea”. Esto se aplica ya sea que la clase derivada necesite o no parámetros propios.

## ¿Cuándo se invocan los constructores?

En la pasada explicación sobre la herencia y la jerarquía de clases, tal vez se te hayan ocurrido dos importantes preguntas: ¿cuándo se crea un objeto de una clase derivada?, ¿cuál constructor se ejecuta primero, el de la clase derivada o el definido en la base? Por ejemplo, dada una clase derivada de nombre **B** y una base llamada **A**, ¿el constructor de **A** es invocado antes que el de **B**, o viceversa? La respuesta es que en una jerarquía de clases, los constructores son invocados en orden de de-

## 312 Fundamentos de C# 3.0

---

rivación, a partir de la clase base y hacia las derivadas. Más aún, este orden es el mismo se utilice o no la palabra clave **base**; en caso de que no, el constructor predeterminado (sin parámetros) de cada clase base será ejecutado. El siguiente programa ilustra este principio:

```
// Muestra el orden de invocación de los constructores.  
using System;  
  
// Crea una clase base.  
class A {  
    public A() {  
        Console.WriteLine("Construye A.");  
    }  
}  
  
// Crea una clase derivada de A.  
class B : A {  
    public B() {  
        Console.WriteLine("Construye B.");  
    }  
}  
  
// Crea una clase derivada de B.  
class C : B {  
    public C() {  
        Console.WriteLine("Construye C.");  
    }  
}  
  
class OrdenDeConstrucción {  
    static void Main() {  
  
        C c = new C();  
    }  
}
```

Los datos de salida generados por el programa son:

```
Construye A  
Construye B  
Construye C
```

Como puedes ver, los constructores son invocados en orden de derivación.

Tiene sentido que los constructores sean invocados en orden de derivación, porque una clase base no tiene conocimiento de ninguna clase derivada, cualquier inicialización que requiera hacer está separada de la inicialización realizada por la derivada, y posiblemente sea un requisito. Por lo mismo, debe ser ejecutada primero.

## Referencias a la clase base y objetos derivados

Como sabes, C# es un lenguaje fuertemente tipificado. Aparte de las conversiones estándar y promociones automáticas que aplican a sus tipos simples, la compatibilidad de tipos es estrictamente forzosa. Por lo mismo, una variable de referencia para un tipo de clase normalmente no puede hacer su referencia a un objeto de otro tipo de clase. Por ejemplo, analiza el siguiente programa:

```
// Esto no compilará.
class X {
    int a;

    public X(int i) { a = i; }

}

class Y {
    int a;

    public Y(int i) { a = i; }

}

class IncompatibleRef {
    static void Main() {
        X x = new X(10);
        X x2;
        Y y = new Y(5);

        x2 = x; // OK, ambos del mismo tipo

        x2 = y; // Error, no es del mismo tipo ← Estas referencias no son compatibles.
    }
}
```

Aquí, aunque la clase **X** y la clase **Y** son estructuralmente la misma, no es posible asignar una referencia de tipo **X** a una variable de tipo **Y**, porque son de tipos diferentes. En general, una variable de referencia puede referirse sólo a objetos del mismo tipo.

Sin embargo, hay una excepción importante al estricto tipo forzoso de C#. Una variable de referencia de una clase base puede asignar una referencia a un objeto de cualquier clase derivada de ella. Esto es legal porque una instancia de un tipo derivado encapsula una instancia del tipo base. Así, una referencia de la base puede referirse a su derivada sin problemas. He aquí un ejemplo:

```
// Una referencia de clase base puede referirse a un objeto de clase
// derivada.
using System;

class X {
    public int a;

    public X(int i) {
        a = i;
    }
}
```

## 314 Fundamentos de C# 3.0

```
class Y : X {
    public int b;

    public Y(int i, int j) : base(j) {
        b = i;
    }
}

class BaseRef {
    static void Main() {
        X x = new X(10);
        X x2;
        Y y = new Y(5, 6);

        x2 = x; // OK, ambas del mismo tipo
        Console.WriteLine("x2.a: " + x2.a);

        x2 = y; // aún OK porque Y es derivada de X
        Console.WriteLine("x2.a: " + x2.a);

        // Las referencias X sólo tienen conocimiento de miembros X.
        x2.a = 19; // OK
        // x2.b = 27; // ¡Error! X no tienen a b como miembro.
    }
}
```

OK porque **Y** es derivada de **X**, por lo que **x2** puede hacer referencia a **y**.

Aquí, **Y** es ahora derivada de **X**; por lo mismo, es permisible que **x2** asigne una referencia a un objeto **Y**.

Es importante comprender que se trata del tipo de dato de la variable de referencia, no del tipo de dato del objeto al que se hace referencia, el que determina cuáles miembros pueden ser accedidos. Esto es, cuando la referencia a un objeto de clase derivada es asignada a una variable de referencia de una clase base, tendrás acceso sólo a aquellas partes del objeto definido por la base. A esto se debe que **x2** no pueda acceder a **b** aun cuando haga referencia a un objeto **Y**. Esto tiene sentido porque la clase base no tiene conocimiento de los elementos que añade la clase derivada. Por ello la última línea de código en el programa anterior está comentada para que no se ejecute.

Aunque la explicación anterior parezca un poco esotérica, tiene algunas aplicaciones prácticas de importancia. Una de ellas se describe a continuación. Otra se explica más tarde en este mismo capítulo, cuando se estudien los métodos virtuales.

Una situación de importancia donde las referencias de clases derivadas son asignadas a variables de clases base se presenta cuando los constructores son invocados en una jerarquía de clases. Como sabes, es común que una clase defina un constructor que tome un objeto de su clase como parámetro. Esto permite que la clase construya una copia de ese objeto. Las clases derivadas de tal clase pueden sacar provecho de esa característica. Por ejemplo, a continuación presentamos constructores de **DosDForma** y **Triángulo** que toman un objeto de su propia clase como parámetro.

```
class DosDForma {  
    // ...  
    // Construye una copia de un objeto DosDForma.  
    public DosDForma(DosDForma ob) {  
        Ancho = ob.Ancho;  
        Alto = ob.Alto;  
    }  
    // ...  
  
class Triángulo : DosDForma {  
    // ...  
    // Construye una copia de un objeto Triángulo.  
    public Triángulo(Triángulo ob) : base(ob) {  
        Estilo = ob.Estilo;  
    }  
    // ...
```

Observa que el constructor **Triángulo** recibe un objeto de tipo **Triángulo**, y transmite ese objeto (a través de **base**) al constructor de **DosDForma**. El punto clave es que **DosDForma()** está esperando un objeto **DosDForma**. Sin embargo, **Triángulo()** transmite un objeto **Triángulo**. Como se explicó, la razón de que esto funcione es porque una referencia de clase base puede referirse a un objeto de clase derivada. Así, es perfectamente aceptable transmitir a **DosDForma()** la referencia de un objeto de una clase derivada de ella misma. Porque el constructor **DosDForma()** inicializa sólo aquellas porciones del objeto de la clase derivada que son miembros de **DosDForma**, sin importar que el objeto pueda contener también otros miembros añadidos por la clase derivada.

## Métodos virtuales y anulación

Un *método virtual* es aquel que se declara como **virtual** en la clase base. Las características que lo identifican es que puede ser redefinido en una o más clases derivadas. De esta manera, cada clase derivada puede tener su propia versión de un método virtual. Los métodos virtuales son interesantes debido a lo que sucede cuando uno de ellos es invocado a través de la referencia a una clase base. En esta situación, C# determina cuál versión del método debe invocar dependiendo del *tipo* del objeto al que se está *haciendo referencia*, y esta determinación se realiza *en tiempo de ejecución*. De esta manera, cuando se hace referencia a varios objetos, se ejecutan las diferentes versiones del método virtual. En otras palabras, es el tipo del objeto al que se hace referencia (no el tipo de la referencia) lo que determina cuál versión del método virtual será ejecutada. Por lo mismo, si una clase base contiene un método virtual y se derivan otras clases de ella, entonces, cuando diferentes tipos de objetos son referidos a través de una referencia a la base, se ejecutan diferentes versiones del método virtual.

Declaras un método como virtual dentro de la clase base antecediendo su declaración con la palabra clave **virtual**. Cuando un método virtual es redefinido por una clase derivada, se utiliza el modificador **override** (anular). Así, el proceso de redefinir un método virtual dentro de una clase derivada es llamado *método de anulación*. Cuando se anula un método, el nombre, tipo de regreso y firma del método que está anulando tienen que ser los mismos que el método virtual que está siendo anulado. Además, un método virtual no puede ser especificado como **static** o **abstract** (se explica más adelante en este mismo capítulo).

## 316 Fundamentos de C# 3.0

La anulación de métodos sienta las bases de uno de los conceptos más poderosos de C#: *envío dinámico de métodos*. El envío dinámico de métodos es el mecanismo a través del cual una invocación a un método anulado es resuelta en tiempo de ejecución en lugar de hacerse en tiempo de compilación. El envío dinámico de métodos es importante porque es la manera como C# implementa el polimorfismo en tiempo de ejecución.

A continuación presentamos un ejemplo que ilustra los métodos virtuales y la anulación:

```
// Muestra el funcionamiento de un método virtual.  
using System;  
  
class Base {  
    // Crea un método virtual en la clase base.  
    public virtual void Quién() { ← Declara un método virtual.  
        Console.WriteLine("Quién() en Base");  
    }  
}  
  
class Derivada1 : Base {  
    // Anula Quién() en una clase derivada.  
    public override void Quién() { ←  
        Console.WriteLine("Quién() en Derivada1");  
    }  
}  
  
class Derivada2 : Base {  
    // Anula Quién() otra vez en otra clase derivada.  
    public override void Quién() { ←  
        Console.WriteLine("Quién() en Derivada2");  
    }  
}  
  
class AnulaDemo {  
    static void Main() {  
        Base baseOb = new Base();  
        Derivada1 dOb1 = new Derivada1();  
        Derivada2 dOb2 = new Derivada2();  
  
        Base baseRef; // una referencia clase base  
  
        baseRef = baseOb;  
        baseRef.Quién(); ←  
  
        baseRef = dOb1;  
        baseRef.Quién(); ←  
  
        baseRef = dOb2;  
        baseRef.Quién(); ←  
    }  
}
```

Anula el método virtual.

En cada caso, la versión de **Quién()** a invocar es determinada en tiempo de ejecución por el tipo de objeto al que se hace referencia.

Los datos generados por este programa son:

```
Quién() en Base
Quién() en Derivada1
Quién() en Derivada2
```

Este programa crea una clase base llamada, precisamente, **Base** y dos clases derivadas de ella, llamadas **Derivada1** y **Derivada2**. **Base** declara un método llamado **Quién()**, y las clases derivadas lo anulan. Los objetos de tipo **Base**, **Derivada1** y **Derivada2** son declarados dentro del método **Main()**. También se declara una referencia de tipo **Base**, llamada **baseRef**. Entonces, el programa asigna una referencia de cada tipo de objeto a **baseRef** y utiliza esa referencia para invocar **Quién()**. Como lo muestran los datos de salida, la versión de **Quién()** que es ejecutada se determina por el tipo de objeto al que se hace referencia en el momento en que se ejecuta la invocación, y no por el tipo de clase de **baseRef**.

No es necesario anular un método virtual. Si una clase derivada no proporciona su propia versión del método virtual, entonces se utiliza el de la clase base. Por ejemplo:

```
/* Cuando un método virtual no se anula, se utiliza
   el método de la clase base. */
using System;

class Base {
    // Crea un método virtual en la clase base.
    public virtual void Quién() {
        Console.WriteLine("Quién() en Base");
    }
}

class Derivada1 : Base {
    // Anula Quién() en una clase derivada.
    public override void Quién() {
        Console.WriteLine("Quién() en Derivada1");
    }
}

class Derivada2 : Base { ← Aquí no se anula Quién().
    // Esta clase no anula Quién().
}

class NoAnulaDemo {
    static void Main() {
        Base baseOb = new Base();
        Derivada1 dOb1 = new Derivada1();
        Derivada2 dOb2 = new Derivada2();

        Base baseRef; // una referencia clase base
        baseRef = baseOb;
        baseRef.Quién();
```

```
baseRef = dObj1;
baseRef.Quién();

baseRef = dObj2;
baseRef.Quién(); // invoca Quién() de la base ←— Invoca Quién() de la base.
}

}
```

Los datos generados por este programa son:

```
Quién() en Base
Quién() en Derivada1
Quién() en Base
```

Aquí, **Derivada2** no anula **Quién()**. Por ello, cuando se invoca **Quién()** en los objetos de **Derivada2**, se ejecuta el **Quién()** de la base.

### Pregunta al experto

**P:** ¿Las propiedades pueden ser virtuales?

**R:** Sí, las propiedades pueden ser modificadas por la palabra clave **virtual** y se pueden anular utilizando la palabra **override**. Lo mismo se aplica a los indexadores.

### ¿Por qué métodos anulados?

Los métodos anulados permiten a C# soportar polimorfismo en tiempo de ejecución. El polimorfismo es esencial en la programación orientada a objetos por una razón: permite que una clase general especifique métodos que serán comunes a todas sus derivadas, mientras que permite que las clases derivadas definan la implementación específica de algunos o todos sus métodos. La anulación de métodos representa otra manera en que C# implementa el aspecto del polimorfismo “una interfaz, múltiples métodos”.

Parte de aplicar exitosamente el polimorfismo es comprender que la clase base y las clases derivadas forman una jerarquía que se mueve de menor a mayor en términos de especialización. Utilizada correctamente, la clase base proporciona todos los elementos que una clase derivada puede utilizar directamente. También define aquellos métodos que la clase derivada debe implementar por sí misma. Esto proporciona a la clase derivada la flexibilidad para definir sus propios métodos, pero aún así la obliga a tener una interfaz consistente. Por ello, gracias a la combinación de herencia con la anulación de métodos, una clase base puede definir el formato general de los métodos que serán utilizados por todas sus clases derivadas.

### Aplicar métodos virtuales

Para comprender mejor el poder de los métodos virtuales, los aplicaremos a la clase **DosDForma**. En los ejemplos anteriores, cada clase derivada de **DosDForma** define un método llamado **Área()**. Esto sugiere que tal vez sea mejor hacer de **Área()** un método virtual de la clase **Dos-**

**DForma**, para permitir que cada clase derivada lo anule y defina cómo será calculada el área para el tipo de forma que encapsula la clase. El siguiente programa lo hace. Por mera conveniencia del ejercicio, también añade una propiedad autoimplementada llamada **Nombre** para **DosDForma**. (Esto facilita la demostración de clases.)

```
// Utiliza métodos virtuales y polimorfismo.
using System;

class DosDForma {
    double pri_ancho;
    double pri_alto;

    // Un constructor predeterminado.
    public DosDForma() {
        Ancho = Alto = 0.0;
        Nombre = "null";
    }

    // Especifica toda la información.
    public DosDForma(double w, double h, string n) {
        Ancho = w;
        Alto = h;
        Nombre = n;
    }

    // Construye objeto con ancho y alto iguales.
    public DosDForma(double x, string n) {
        Ancho = Alto = x;
        Nombre = n;
    }

    // Construye una copia de objeto DosDForma.
    public DosDForma(DosDForma ob) {
        Ancho = ob.Ancho;
        Alto = ob.Alto;
        Nombre = ob.Nombre;
    }

    // Propiedades para Ancho, Alto y Nombre.
    public double Ancho {
        get { return pri_ancho; }
        set { pri_ancho = value < 0 ? -value : value; }
    }

    public double Alto {
        get { return pri_alto; }
        set { pri_alto = value < 0 ? -value : value; }
    }

    public string Nombre { get; set; }
```

```
public void MuestraDim() {
    Console.WriteLine("Ancho y Alto son " +
                      Ancho + " y " + Alto);
}

public virtual double Area() {←———— Ahora Área() es virtual.
    Console.WriteLine("Área() debe ser anulada.");
    return 0.0;
}
}

// Una clase derivada de DosDForma para triángulos.
class Triángulo : DosDForma {
    string Estilo;

    // Un constructor predeterminado.
    public Triángulo() {
        Estilo = "null";
    }

    // Constructor que toma estilo, ancho y alto.
    public Triángulo(string s, double w, double h) :
        base(w, h, "triángulo") {
        Estilo = s;
    }

    // Construye un triángulo isósceles.
    public Triángulo(double x) : base(x, "triángulo") {
        Estilo = "isósceles";
    }

    // Construye una copia de un objeto Triángulo.
    public Triángulo(Triángulo ob) : base(ob) {
        Estilo = ob.Estilo;
    }

    // Anula Área() para Triángulo.
    public override double Area() {←———— Anula Área() para Triángulo.
        return Ancho * Alto / 2;
    }

    public void MuestraEstilo() {
        Console.WriteLine("Triángulo es " + Estilo);
    }
}

// Una clase derivada de DosDForma para rectángulos.
class Rectángulo : DosDForma {
```

```

// Constructor que toma ancho y alto.
public Rectángulo(double w, double h) :
    base(w, h, "rectángulo") { }

// Construye un cuadrado.
public Rectángulo(double x) :
    base(x, "rectángulo") { }

// Construye un objeto a partir de otro.
public Rectángulo(Rectángulo ob) : base(ob) { }

// Regresa true si el rectángulo es un cuadrado.
public bool EsCuadrado() {
    if (Ancho == Alto) return true;
    return false;
}

// Anula Área() para Rectángulo.
public override double Área() { ← Anula Área() para Rectángulo.
    return Ancho * Alto;
}
}

class FormasDin {
    static void Main() {
        DosDForma[] formas = new DosDForma[5];

        formas[0] = new Triángulo("recto", 8.0, 12.0);
        formas[1] = new Rectángulo(10);
        formas[2] = new Rectángulo(10, 4);
        formas[3] = new Triángulo(7.0);
        formas[4] = new DosDForma(10, 20, "genérico");

        for(int i=0; i < formas.Length; i++) {
            Console.WriteLine("objeto es " + formas[i].Nombre);
            Console.WriteLine("Área es " + formas[i].Área());
            ↑
            Console.WriteLine();
        }
    }
}

```

↑  
La versión apropiada de Área()  
es invocada para cada forma.

Los datos de salida generados por el programa son:

```
objeto es triángulo
Área es 48
```

```
objeto es rectángulo
```

```
Área es 100
objeto es rectángulo
Área es 40
objeto es triángulo
Área es 24.5
objeto es genérico
Área() debe ser anulada
Área es 0
```

Examinemos de cerca este programa. Primero, como se explicó, `Área()` se declara como **virtual** en la clase **DosDForma** y es anulada por **Triángulo** y **Rectángulo**. Dentro de **DosDForma**, `Área()` recibe una implementación de marcador de posición que simplemente informa al usuario que este método debe ser anulado por una clase derivada. Cada anulación de `Área()` proporciona una implementación que se puede acoplar al tipo de objeto encapsulado por la clase derivada. Así, si fueras a implementar una clase elipse, por ejemplo, entonces `Área()` tendría que calcular el área de una elipse.

Hay una característica de importancia adicional en el programa anterior. Observa en `Main()` que **formas** es declarada como un arreglo de objetos **DosDForma**. Sin embargo, a los elementos de este arreglo se les asignan referencias **Triángulo**, **Rectángulo** y **DosDForma**. Esto es válido porque una referencia de clase base puede hacer referencia al objeto de una clase derivada. Entonces, el programa realiza ciclos repetitivos a través del arreglo, mostrando información sobre cada objeto. Aunque muy sencillo, este programa ilustra el poder de la herencia y la anulación de métodos. El tipo de objetos almacenados en una variable de referencia de una clase base se determina en tiempo de ejecución y actúa en consecuencia. Si un objeto es derivado de **DosDForma**, entonces su área puede ser obtenida invocando `Área()`. La interfaz para esta operación es la misma, no importa qué tipo de forma se esté utilizando.

## Utilizar clases abstractas

En algunas ocasiones necesitarás crear una clase base que defina sólo un formato generalizado que será compartido por todas sus clases derivadas, dejando a estas últimas la tarea de llenar los detalles. Una clase así determina la naturaleza de los métodos que las clases derivadas deben implementar, pero no proporciona en sí la implementación de uno o más de esos métodos. Esta situación puede ocurrir cuando una clase base no es capaz de crear una implementación que tenga significado para un método. Éste es el caso con la versión de **DosDForma** utilizada en el ejemplo anterior. La definición de `Área()` es simplemente un marcador de posición. No calculará ni mostrará el área de ningún tipo de objeto.

Como verás cuando crees tus propias bibliotecas, es común que un método carezca de definición significativa en el contexto de su clase base. Puedes manejar esta situación de dos maneras. La primera, como se muestra en el ejemplo anterior, es simplemente dejar que se produzca el mensaje de advertencia. Este enfoque puede ser útil en algunas situaciones, como en la depuración, pero no es lo más apropiado. Puedes tener métodos que *necesitan* ser anulados por la clase derivada con el fin de que ésta tenga un significado. Considera la clase **Triángulo**. No tiene significado si `Área()` no se define. En este caso, querrás una manera para asegurar que una clase derivada anule, en efecto, todos los métodos necesarios. La solución que ofrece C# a este problema es el *método abstracto*.

Un método abstracto se crea especificando el modificador de tipo **abstract**. Un método abstracto no tiene cuerpo y, por lo mismo, no es implementado por la clase base. Así, una clase derivada debe anularlo, porque no puede utilizar la versión definida en la base. Como posiblemente lo adivines, un método abstracto es automáticamente virtual y no hay necesidad de utilizar el modificador **virtual**. De hecho, es un error utilizar **virtual** y **abstract** juntos.

Para declarar un método abstracto, utiliza el siguiente formato:

```
abstract nombre tipo(lista-parámetros);
```

Como puedes ver, no está presente ningún cuerpo de método. El modificador **abstract** puede ser utilizado únicamente en métodos de instancia. No puede ser aplicado a métodos **static**.

Una clase que contiene uno o más métodos abstractos también debe ser declarada abstracta utilizando el especificador **abstract** antes de la declaración **class**. Como una clase abstracta no define una implementación completa, no puede haber objetos de la misma. Por lo mismo, intentar la creación de un objeto a partir de una clase abstracta utilizando la palabra clave **new** traerá como resultado un error en tiempo de compilación.

Cuando una clase derivada hereda una clase abstracta, debe implementar todos los métodos abstractos de la clase base. De no hacerlo, la derivada también debe especificarse como **abstract**. De esta manera, el atributo **abstract** se hereda hasta el momento en que se lleva a cabo la implementación.

Puedes mejorar la clase **DosDForma** utilizando una clase abstracta. Como no hay un concepto significativo de área para una figura bidimensional indefinida, la siguiente versión del programa anterior declara **Área()** como **abstract** dentro de **DosDForma** y también declara esta última como **abstract**. Por supuesto, esto significa que todas las clases derivadas de **DosDForma** deben anular **Área()**.

```
// Crea una clase abstracta.
using System;

abstract class DosDForma { ←———— Ahora DosDForma es abstracto.
    double pri_ancho;
    double pri_alto;

    // Un constructor predeterminado.
    public DosDForma() {
        Ancho = Alto = 0.0;
        Nombre = "null";
    }

    // Parametriza constructor.
    public DosDForma(double w, double h, string n) {
        Ancho = w;
        Alto = h;
        Nombre = n;
    }

    // Construye objeto con ancho y alto iguales.
    public DosDForma(double x, string n) {
        Ancho = Alto = x;
        Nombre = n;
    }
}
```

```
// Construye un objeto a partir de otro.
public DosDForma(DosDForma ob) {
    Ancho = ob.Ancho;
    Alto = ob.Alto;
    Nombre = ob.Nombre;
}

// Propiedades de Ancho, Alto y Nombre.
public double Ancho {
    get { return pri_ancho; }
    set { pri_ancho = value < 0 ? -value : value; }
}

public double Alto{
    get { return pri_alto; }
    set { pri_alto = value < 0 ? -value : value; }
}

public string Nombre { get; set; }

public void MuestraDim() {
    Console.WriteLine("Ancho y alto son " +
                      Ancho + " y " + Alto);
}

// Ahora, Área() es abstracto.
public abstract double Area();————— Ahora Área() es abstracto.
}

// Una clase derivada de DosDForma para triángulos.
class Triángulo : DosDForma {
    string Estilo;

    // Un constructor predeterminado.
    public Triángulo() {
        Estilo = "null";
    }

    // Constructor que toma estilo, ancho y alto.
    public Triángulo(string s, double w, double h) :
        base(w, h, "triángulo") {
        Estilo = s;
    }

    // Construye un triángulo isósceles.
    public Triángulo(double x) : base(x, "triángulo") {
        Estilo = "isósceles";
    }
}
```

```
// Construye un objeto a partir de otro.
public Triángulo(Triángulo ob) : base(ob) {
    Estilo = ob.Estoilo;
}

// Anula Área() para Triángulo.
public override double Área() { ←———— Anula Área() para Triángulo.
    return Ancho * Alto / 2;
}

public void MuestraEstilo() {
    Console.WriteLine("Triángulo es " + Estilo);
}

// Una clase derivada de DosDForma para rectángulos.
class Rectángulo : DosDForma {
    // Constructor que toma ancho y alto.
    public Rectángulo(double w, double h) :
        base(w, h, "rectángulo") { }

    // Construye un cuadrado.
    public Rectángulo(double x) :
        base(x, "rectángulo") { }

    // Construye un objeto a partir de otro.
    public Rectángulo(Rectángulo ob) : base(ob) { }

    // Regresa true si el rectángulo es un cuadrado.
    public bool EsCuadrado() {
        if (Ancho == Alto) return true;
        return false;
    }

    // Anula Área() para Rectángulo.
    public override double Área() { ←———— Anula Área() para Rectángulo.
        return Ancho * Alto;
    }
}

class AbsForma {
    static void Main() {
        DosDForma[] formas = new DosDForma[4];

        formas[0] = new Triángulo("recto", 8.0, 12.0);
        formas[1] = new Rectángulo(10);
        formas[2] = new Rectángulo(10, 4);
        formas[3] = new Triángulo(7.0);
```

```
        for(int i=0; i < formas.Length; i++) {
            Console.WriteLine("objeto es " + formas[i].Nombre);
            Console.WriteLine("Área es " + formas[i].Área());

            Console.WriteLine();
        }
    }
}
```

Como lo ilustra el programa, todas las clases derivadas *deben* anular `Área()` (o ser declaradas también **abstractas**). Para que lo compruebes por ti mismo, intenta crear una clase derivada que no anule `Área()`. Recibirás un error en tiempo de compilación. Por supuesto, todavía es posible crear una referencia de objeto de tipo **DosDForma**, lo cual hace el programa. Sin embargo, ya no es posible declarar objetos del tipo **DosDForma**. Debido a ello, en `Main()`, el arreglo **formas** ha sido reducido a cuatro y ya no se crea un objeto **DosDForma**.

Un último punto: observa que **DosDForma** aún incluye el método `MuestraDim()` y que no es modificado por **abstract**. Es perfectamente aceptable (de hecho es muy común), que una clase abstracta contenga métodos concretos que una clase derivada es libre de utilizar tal y como son. Sólo aquellos métodos declarados como **abstractos** deben ser anulados por las clases derivadas.

## Utilizar **sealed** para prevenir la herencia

A pesar de lo poderosa y útil que es la herencia, en algunas ocasiones tendrás que prevenirla. Por ejemplo, puedes tener una clase que encapsule la secuencia de inicialización de un dispositivo de hardware especializado, como un monitor médico. En este caso, no querrás que los usuarios de tu clase puedan cambiar la manera en que se inicializa el monitor, porque es posible que configuren de manera incorrecta el dispositivo. Sea cual sea la razón, en C# es fácil prevenir la herencia de una clase utilizando la palabra clave **sealed** (sellado).

Para prevenir que una clase sea heredada, antecede su declaración con **sealed**. Como era de esperarse, es ilegal declarar una clase como **abstract** y **sealed**, ya que una clase abstracta es incompleta en sí y depende de sus clases derivadas para proporcionar implementaciones completas.

He aquí un ejemplo de clase **sealed**:

```
sealed class A { ← Esta clase no puede ser heredada.
    // ...
}

// La clase siguiente es ilegal.
class B : A { // !ERROR! No se puede derivar la clase A
    // ...
}
```

Como lo indican los comentarios, es ilegal que **B** herede a **A**, porque **A** es declarada **sealed**.

Otro punto: **sealed** también puede ser utilizada en métodos virtuales para prevenir anulaciones posteriores. Por ejemplo, supongamos que existe una clase base llamada **B** y una clase derivada llamada **D**. Un método declarado **virtual** en **B** puede ser declarado **sealed** en **D**. Esto prevendría

que cualquier otra clase que herede **D** anule el método. Esta situación se ilustra en el siguiente ejemplo:

```
class B {
    public virtual void MiMétodo() { /* ... */ }
}

class D : B {
    // Esto sella MiMétodo() y previene posteriores anulaciones.
    sealed public override void MiMétodo() { /* ... */ }
}

class X : D {                                Ahora, este método no puede ser anulado.
    // ¡Error! ¡MiMétodo() está sellado!
    public override void MiMétodo() { /* ... */ }
}
```



Como **MiMétodo()** está sellado por **D**, no puede ser anulado por **X**.

## La clase **object**

C# define una clase especial llamada **object** (objeto) que es una clase base implícita para todas las demás clases y para todos los demás tipos (incluyendo tipos de valor). En otras palabras, todos los tipos C# derivan de **object**. Esto significa que una variable de referencia de tipo **object** puede hacer referencia a un objeto de cualquier otro tipo. Además, como los arreglos se implementan como objetos, una variable de tipo **object** también puede hacer referencia a cualquier arreglo. Técnicamente, el nombre **object** de C# es sólo otro nombre de **System.Object**, que es parte de la biblioteca de clases.NET Framework.

La clase **object** define los siguientes métodos, lo que significa que están disponibles en todos los objetos.

Método	Propósito
public virtual bool Equals(object ob)	Determina si el objeto invocador es el mismo referido por <i>ob</i> .
public static bool Equals(object ob1, object ob2)	Determina si <i>ob1</i> es el mismo que <i>ob2</i> .
protected virtual Finalize( )	Realiza acciones de cierre antes de la recolección de basura. En C#, <b>Finalize</b> es accesado a través de un destructor.
public virtual int GetHashCode( )	Regresa el código hash asociado con el objeto invocado.
public type GetType( )	Obtiene el tipo de un objeto en tiempo de ejecución.
protected object MemberwiseClone( )	Hace una “copia somera” de un objeto. Se trata de una acción en la que los miembros son copiados, pero no así los objetos referenciados por los miembros.
public static bool ReferenceEquals(object ob1, object ob2)	Determina si <i>ob1</i> y <i>ob2</i> hacen referencia al mismo objeto.
public virtual string ToString( )	Regresa una cadena de caracteres que describen al objeto.

Pocos de estos métodos requieren explicación adicional. De forma predeterminada, el método **Equals(object)** determina si el objeto invocador hace referencia al mismo objeto al que hace referencia el argumento. Es decir, determina si dos referencias son la misma. Regresa **true** si los objetos son el mismo y **false** de lo contrario. Puedes anular este método en las clases que tú creas. Eso te permite definir los medios de comparación relativos a la clase. Por ejemplo, podrías definir **Equals(object)** para que comparara el contenido de dos objetos buscando igualdades. El método **Equals(object, object)** invoca **Equals(object)** para calcular el resultado.

El método **GetHashCode()** regresa un código hash asociado con el objeto invocador. Este código hash puede ser utilizado en cualquier algoritmo que ocupe hash como medio para accesar objetos almacenados.

Como se mencionó en el capítulo 7, si sobrecargas el operador **==**, por lo regular necesitas anular **Equals(object)** y **GetHashCode()**, porque la mayoría de las veces querrás que el operador **==** y que el método **Equals(object)** funcionen de la misma manera. Cuando se anula **Equals()**, también es necesario anular **GetHashCode()**, para que los dos métodos sean compatibles.

El método **ToString()** regresa una cadena de caracteres que contiene una descripción del objeto sobre el cual se aplica el método. Además, este método es invocado en automático cuando un objeto es mostrado utilizando **WriteLine()**. Muchas clases anulan este método. Eso les permite crear una descripción personalizada y específica para el tipo de objeto que crean. Por ejemplo:

```
// Muestra el funcionamiento de ToString()
using System;

class MiClase {
    static int count = 0;
    int id;

    public MiClase() {
        id = count;
        count++;
    }

    public override string ToString() { ← Anula ToString().
        return "MiClase objeto #" + id;
    }
}

class Test {
    static void Main() {
        MiClase ob1 = new MiClase();
        MiClase ob2 = new MiClase();
        MiClase ob3 = new MiClase();

        Console.WriteLine(ob1);
        Console.WriteLine(ob2); ← Aquí, ToString() es invocado en automático.
        Console.WriteLine(ob3);
    }
}
```

Los datos generados por este programa son:

```
MiClase objeto #0
MiClase objeto #1
MiClase objeto #2
```

## Encajonar y desencajonar

Como se explicó, todos los tipos C#, incluyendo los tipos valor, se derivan de **object**. Así, una referencia de tipo **object** puede ser utilizada para referirse a cualquier otro tipo, incluyendo los de valor. Cuando una referencia **object** se refiere a un tipo valor, ocurre lo que se conoce como *encajonar*. El encajonamiento provoca que la cantidad contenida en un tipo valor sea almacenada en una instancia de objeto. Así, un tipo valor es “encajonado” dentro de un objeto. Este objeto puede ser utilizado como cualquier otro. En todos los casos, el encajonamiento ocurre de manera automática. El programador simplemente asigna un valor a una referencia **object**. C# se ocupa del resto.

*Desencajonar* es el proceso de recuperar un valor de un objeto. Esta acción se realiza utilizando una transformación de la referencia **object** al tipo valor deseado. Si intentas desencajonar un objeto a un tipo incompatible dará como resultado un error en tiempo de ejecución.

He aquí un ejemplo que ilustra las acciones de encajonar y desencajonar:

```
// Un ejemplo sencillo de encajonar/desencajonar.
using System;

class EncajonaDemo {
    static void Main() {
        int x;
        object obj;

        x = 10;
        obj = x; // encajona x en un objeto ← Aquí, el valor de x se encajona.

        int y = (int)obj; // desencajona obj en un int ← Aquí, el valor se
        Console.WriteLine(y);
    }
}
```

Este programa muestra el valor 10. Observa que el valor en **x** es encajonado simplemente al asignarlo a **obj**, que es una referencia de **objeto**. El valor entero en **obj** se recupera al transformar **obj** en **int**.

A continuación presentamos un ejemplo más interesante de encajonamiento. En este caso, un **int** se transmite como argumento a un método **Sqr()** que calcula su cuadrado utilizando un **object** como argumento.

```
// El encajonamiento también ocurre cuando se transmiten valores.
using System;

class EncajonaDemo {
    static void Main() {
        int x;

        x = 10;
        Console.WriteLine("Ésta es x: " + x);
```

```
// x es automáticamente encajonada cuando pasa a Sqr().  
x = EncajonaDemo.Sqr(x); ← El valor de x es  
Console.WriteLine("Éste es el cuadrado de x: " + x); encajonado cuando  
} Sqr() es invocado.  
  
static int Sqr(object o) {  
    return (int)o * (int)o;  
}  
}
```

Los datos generados por este programa son:

```
Ésta es x: 10  
Éste es el cuadrado de x: 100
```

Aquí, el valor de x es encajonado automáticamente cuando se transmite a **Sqr()**.

El encajonamiento y desencajonamiento permiten que el sistema de tipos de C# sea completamente unificado. Todos los tipos derivan de **object**. Una referencia a cualquier tipo puede ser asignada a una variable de tipo **object**. Encajar/Desencajar maneja en automático los detalles de los tipos valor. Más aún, como todos los tipos son derivados de **object**, todos ellos tienen acceso a los métodos de **object**. Por ejemplo, analiza el siguiente programa, que es algo sorprendente:

```
// ¡Encajonar hace posible invocar métodos sobre un valor!  
using System;  
  
class MétodoValor {  
    static void Main() {  
  
        Console.WriteLine(186.ToString()); ← ¡Perfectamente legal en C#!  
    }  
}
```

Este programa muestra en pantalla el número 186. La razón es que el método **ToString()** regresa una representación en cadena de caracteres del objeto sobre el cual fue invocado. En este caso, la representación en cadena de caracteres de 186 ¡es 186!

### ✓ Autoexamen Capítulo 8

1. ¿Una clase base tiene acceso a los miembros de una clase derivada? ¿Una clase derivada tiene acceso a los miembros de una clase base?
2. Crea una clase derivada de **DosDForma** llamada **Círculo**. Incluye un método **Área()** que calcule el área de un círculo y un constructor que utilice **base** para inicializar la porción de **DosDForma**.
3. ¿Cómo previenes que una clase derivada tenga acceso a un miembro de una clase base?

- 4.** Describe el propósito de **base**.
  - 5.** Dada la siguiente jerarquía, ¿en qué orden son invocados los constructores de estas clases cuando se inicializa un objeto **Gamma**?
- ```
class Alpha { ...  
class Beta : Alpha { ...  
class Gamma : Beta { ...
```
- 6.** Una referencia de clase base puede hacer referencia a un objeto de clase derivada. Explica por qué es una característica importante en relación con la anulación de métodos.
  - 7.** ¿Qué es una clase abstracta?
  - 8.** ¿Cómo evitas que una clase sea heredada?
  - 9.** Explica cómo se utilizan la herencia, la anulación de métodos y las clases abstractas para soportar el polimorfismo.
  - 10.** ¿Qué clase es base para el resto de las clases?
  - 11.** Explica el encajonado.
  - 12.** ¿Cómo pueden ser accesados los miembros **protected**?



# Capítulo 9

Interfaces, estructuras  
y enumeraciones

## Habilidades y conceptos clave

- Fundamentos de la interfaz
  - Añadir propiedades e indexadores a la interfaz
  - Heredar interfaces
  - Implementaciones explícitas
  - Estructuras
  - Enumeraciones
- 

Este capítulo trata una de las características más importantes de C#: la interfaz. Una *interfaz* define un conjunto de métodos que serán implementados por una clase. Una interfaz no implementa ningún método por sí misma. Se trata de una construcción puramente lógica que describe un conjunto de métodos que proporcionará una clase sin dictar detalles específicos de implementación.

También se abordan en este capítulo otros dos tipos de datos: *estructuras* y *enumeraciones*. Las estructuras son similares a las clases, excepto que son manejadas como tipos valor en lugar de tipos referencia. Las enumeraciones son listas de constantes de entero con nombre. Estructuras y enumeraciones contribuyen al enriquecimiento del ambiente de programación de C#, proporcionando soluciones elegantes a muchos problemas de cómputo.

## Interfaces

En la programación orientada a objetos, en ocasiones es de ayuda definir lo que una clase debe hacer, pero no cómo debe hacerlo. Ya viste un ejemplo de esto: el método abstracto. El método abstracto define la firma de un método, pero no proporciona su implementación. Una clase derivada debe proporcionar su propia implementación para cada método abstracto definido por su clase base. De esta manera, un método abstracto especifica la *interfaz* para el método, pero no la *implementación*. Aunque las clases y los métodos abstractos son de utilidad, es posible llevar un paso adelante este concepto. En C#, puedes separar por completo la interfaz de la clase de su implementación utilizando la palabra clave **interface**.

Las interfaces son sintácticamente parecidas a las clases abstractas. Sin embargo, en una interfaz, ningún método puede tener cuerpo. Es decir, una interfaz no proporciona ningún tipo de implementación. Especifica lo que se debe realizar, pero no cómo debe realizarse. Una vez que se define una interfaz, cualquier cantidad de clases pueden implementarla. De la misma manera, una sola clase puede implementar cualquier cantidad de interfaces.

Para implementar una interfaz, la clase debe proporcionar los cuerpos (implementaciones) para los métodos descritos por la primera. Cada clase es libre de determinar los detalles de sus propias implementaciones. Así, dos clases pueden implementar la misma interfaz de diferentes maneras, pero aun así, cada clase tiene que proporcionar el mismo conjunto de métodos. Por ello,

el código que tiene conocimiento de la interfaz puede utilizar objetos de cualquier clase porque la interfaz para esos objetos es la misma. Al proporcionar la interfaz, C# te permite utilizar por completo el aspecto del polimorfismo “una interfaz, múltiples métodos”.

Las interfaces se declaran utilizando la palabra clave **interface**. He aquí una forma simplificada de una declaración de interfaz:

```
interface nombre {
    tipo-ret nombre-método1(lista-parámetros);
    tipo-ret nombre-método2(lista-parámetros);
    // ...
    tipo-ret nombre-métodoN(lista-parámetros);
}
```

El nombre de la interfaz se especifica en *nombre*. Los métodos se declaran utilizando sólo su tipo de regreso y firma. Son, en esencia, métodos abstractos. Como se explicó, en una interfaz ningún método puede tener implementación. Así, cada clase que incluya la interfaz debe implementar todos los métodos declarados por ella. En una interfaz, los métodos son **public** de manera implícita y no se permite el uso de especificadores de acceso explícitos.

A continuación presentamos un ejemplo de **interface**. Especifica la interfaz a una clase que genera una serie de números.

```
public interface ISeries {
    int GetNext(); // regresa el siguiente número de la serie
    void Reset(); // reinicia
    void SetStart(int x); // establece el valor de inicio
}
```

La interfaz está declarada **public** para que pueda ser implementada por cualquier clase en el programa.

Además de firmas de método, las interfaces pueden declarar firmas de propiedades, indexadores y eventos. Los eventos se describen en el capítulo 12; aquí sólo nos ocuparemos de métodos, propiedades e indexadores. Las interfaces no pueden tener miembros de datos. No pueden definir constructores, destructores ni operadores de métodos. De igual manera, ninguno de sus miembros puede ser declarado como **static**.

## Implementar interfaces

Una vez que **interface** ha sido definida, una o más clases pueden implementarla. Para hacerlo, el nombre de la interfaz se especifica después del nombre de la clase de la misma manera como se hace con una clase base. El formato general de una clase que implementa una interfaz es el siguiente:

```
class nombre-clase : nombre-interfaz {
    // cuerpo de la clase
}
```

El nombre de la interfaz que será implementada se especifica en *nombre-interfaz*.

Cuando una clase implementa una interfaz, debe hacerlo por completo. No puede escoger ni seleccionar cuáles partes implementará.

Las clases pueden implementar más de una interfaz. Para hacerlo, las interfaces se separan con comas. Una clase puede heredar una clase base e implementar una o más interfaces. En este caso, el nombre de la base debe encabezar la lista separada por comas.

Los métodos que implementan una interfaz deben ser declarados **public**. Esto se debe a que los métodos son implícitamente públicos en el interior de una interfaz, por lo que su implementación debe actuar en consecuencia. También, la firma de tipo del método implementado debe coincidir exactamente con la firma de tipo especificada en la definición **interface**.

Aquí presentamos un ejemplo que implementa la interfaz **ISeries** mostrada con anterioridad. Crea una clase llamada **MásDos**,\* que genera una serie de números; cada uno suma dos al anterior.

```
// Implementa ISeries.
class MásDos : ISeries { ←———— Implementa la interfaz ISeries.
    int inicio;
    int val;

    public MásDos() {
        inicio = 0;
        val = 0;
    }

    public int GetNext() {
        val += 2;
        return val;
    }

    public void Reset() {
        inicio = 0;
        val = 0;
    }

    public void SetStart(int x) {
        inicio = x;
        val = x;
    }
}
```

Como puedes ver, **MásDos** implementa los tres métodos definidos por **ISeries**. Como se explicó, esto es necesario porque una clase no puede crear una implementación parcial de la interfaz.

He aquí una clase que muestra el funcionamiento de **MásDos**:

```
// Muestra el funcionamiento de la interfaz MásDos.
using System;

class ISeriesDemo {
    static void Main() {
        MásDos ob = new MásDos();
```

---

\* El uso de acentos y eñes puede generar problemas en algunas situaciones. En los ejemplos que aparecen en el cuerpo del libro se utilizan acentos y eñes por cuestión de ortografía, pero en los archivos de código se eliminan.

```
for(int i=0; i < 5; i++)
    Console.WriteLine("Próximo valor es " +
                      ob.GetNext());

Console.WriteLine("\nResetea");
ob.Reset();
for(int i=0; i < 5; i++)
    Console.WriteLine("Próximo valor es " +
                      ob.GetNext());

Console.WriteLine("\nComienza en 100");
ob.SetStart(100);
for(int i=0; i < 5; i++)
    Console.WriteLine("Próximo valor es " +
                      ob.GetNext());
}
}
```

Para compilar **ISeriesDemo**, debes incluir las clases **ISeries**, **MásDos** e **ISeriesDemo** en la compilación. Se compilarán en automático los tres archivos para crear el ejecutable final. Por ejemplo, si llamas a estos archivos **ISeries.cs**, **MásDos.cs** e **IseriesDemo.cs**, la siguiente línea de comando compilará el programa:

```
>csc ISeries.cs MásDos.cs ISeriesDemo.cs
```

Si estás utilizando el IDE de Visual Studio, simplemente añade los tres archivos a tu proyecto. Otro punto: también es perfectamente válido poner estas tres clases en el mismo archivo.

Los datos de salida generados por el programa son:

```
Próximo valor es 2
Próximo valor es 4
Próximo valor es 6
Próximo valor es 8
Próximo valor es 10
```

```
Reinicia
Próximo valor es 2
Próximo valor es 4
Próximo valor es 6
Próximo valor es 8
Próximo valor es 10
```

```
Comienza en 100
Próximo valor es 102
Próximo valor es 104
Próximo valor es 106
Próximo valor es 108
Próximo valor es 110
```

Es permisible y común que las clases implementen interfaces para definir miembros adicionales propios. Por ejemplo, la siguiente versión de **MásDos** añade el método **GetPrevious()**, que regresa el valor anterior:

```
// Implementa ISeries y GetPrevious().
class MásDos : ISeries {
    int inicio;
    int val;
    int prev;

    public MásDos() {
        inicio = 0;
        val = 0;
        prev = -2;
    }

    public int GetNext() {
        prev = val;
        val += 2;
        return val;
    }

    public void Reset() {
        inicio = 0;
        val = 0;
        prev = -2;
    }

    public void SetStart(int x) {
        inicio = x;
        val = x;
        prev = x - 2;
    }

    // Un método no especificado por ISeries.
    int GetPrevious() { ← Añade un método no especificado por ISeries.
        return prev;
    }
}
```

Observa que la adición de **GetPrevious()** requiere un cambio a las implementaciones de los métodos definidos por **ISeries**. Sin embargo, como la interfaz de esos métodos es la misma, el cambio es transparente y no rompe con el código preexistente. Ésta es una de las ventajas de las interfaces.

Como se explicó, cualquier cantidad de clases pueden implementar una **interface**. Por ejemplo, aquí está una clase llamada **MásTres** que genera una serie de números, cada uno de los cuales suma tres al anterior:

```
// Implementa ISeries.
class MásTres : ISeries { ← Implementa ISeries de manera diferente.
    int inicio;
    int val;
```

```

public MásTres() {
    inicio = 0;
    val = 0;
}

public int GetNext() {
    val += 3;
    return val;
}

public void Reset() {
    inicio = 0;
    val = 0;
}

public void SetStart(int x) {
    inicio = x;
    val = x;
}
}

```

## Utilizar referencias de interfaz

Tal vez te sorprenda saber que puedes declarar una variable de referencia de un tipo interfaz. En otras palabras, puedes crear una variable de referencia de interfaz. Tal variable puede hacer referencia a cualquier objeto que implemente su interfaz. Cuando invocas un método en un objeto a través de una referencia de interfaz, lo que se ejecuta es la versión del método implementado por ese objeto. Este proceso es semejante a utilizar una referencia de clase base para accesar un objeto de clase derivada, como se explicó en el capítulo 8.

El siguiente ejemplo ilustra el uso de una referencia de interfaz. Utiliza la misma variable de referencia para invocar métodos en objetos de **MásDos** y **MásTres**. Por cuestiones de claridad, muestra todas las piezas ensambladas en un solo archivo.

```

// Muestra referencias de interfaz.
using System;

// Define la interfaz.
public interface ISeries {
    int GetNext(); // regresa el siguiente número de la serie
    void Reset(); // reinicia
    void SetStart(int x); // establece punto de inicio
}

// Implementa ISeries de una manera.
class MásDos : ISeries {
    int inicio;
    int val;
}

```

```
public MásDos() {
    inicio = 0;
    val = 0;
}

public int GetNext() {
    val += 2;
    return val;
}

public void Reset() {
    inicio = 0;
    val = 0;
}

public void SetStart(int x) {
    inicio = x;
    val = x;
}

// Implementa ISeries de otra manera.
class MásTres : ISeries {
    int inicio;
    int val;

    public MásTres() {
        inicio = 0;
        val = 0;
    }

    public int GetNext() {
        val += 3;
        return val;
    }

    public void Reset() {
        inicio = 0;
        val = 0;
    }

    public void SetStart(int x) {
        inicio = x;
        val = x;
    }
}
```

```

class ISeriesDemo2 {
    static void Main() {
        MásDos dosOb = new MásDos();
        MásTres tresOb = new MásTres();
        ISeries ob; ← Declarar una variable de referencia de interfaz.

        for(int i=0; i < 5; i++) {
            ob = dosOb;
            Console.WriteLine("Siguiente valor MásDos es " +
                ob.GetNext()); ← Accesa un objeto a
            ob = tresOb;
            Console.WriteLine("Siguiente valor MásTres es " +
                ob.GetNext()); ← través de una refe-
        }
    }
}

```

En **Main()**, **ob** es declarado para ser referencia a una interfaz **ISeries**. Esto significa que puede utilizarse para almacenar referencias a cualquier objeto que implemente **ISeries**. En este caso, es utilizado para referirse a **dosOb** y **tresOb**, que son objetos de tipo **MásDos** y **MásTres**, respectivamente, y ambos implementan **ISeries**. Una variable de referencia de interfaz sólo conoce los métodos mencionados en su declaración **interface**. Así, **ob** no puede utilizarse para accesar otras variables o métodos que tal vez soporte el objeto.

**Prueba esto****Crear una interfaz de orden en cola**

Para ver el poder de las interfaces en acción, veremos un ejemplo práctico. En capítulos anteriores desarrollaste una clase llamada **ColaSimple** que implementaba un orden en cola sencillo y fijo aplicado a caracteres. Sin embargo, hay muchas maneras de implementar un orden de ese tipo. Por ejemplo, la cola puede ser de tamaño fijo o puede ser “incrementable”. La cola puede ser *lineal*, en cuyo caso puede agotarse, o puede ser *circular*, en cuyo caso los elementos pueden ser añadidos siempre y cuando otros elementos vayan saliendo. La cola también puede estar contenida en un arreglo, en una lista vinculada, en un árbol binario, entre otros. No importa cómo se implemente la cola, su interfaz siempre es la misma, y los métodos **Put()** y **Get()** definen la interfaz de la cola independientemente de los detalles de su implementación. Dado que la interfaz del orden en cola está separada de su implementación, es fácil definir una interfaz de cola, dejando que cada implementación defina los detalles específicos.

Aquí crearemos una interfaz para ordenar caracteres en cola y tres implementaciones. Las tres implementaciones utilizarán un arreglo para almacenar los caracteres. Una cola será la clase **ColaSimple** desarrollada con anterioridad. Otra será una cola circular. En éstas, cuando se llega al final del arreglo subyacente, los índices *get* y *put* regresan al principio automáticamente. De esta manera, se pueden almacenar cualquier número de elementos en una cola circular, siempre y cuando otros elementos vayan saliendo. La implementación final crea una cola dinámica, que crece lo necesario cuando el tamaño es excedido.

(continúa)

## Paso a paso

1. Crea un archivo llamado **ICharQ.cs** y coloca en él la siguiente definición de interfaz:

```
// Una interfaz de cola para caracteres.  
public interface ICharQ {  
    // Coloca un carácter en la cola.  
    void Put(char ch);  
  
    // Obtiene un carácter de la cola.  
    char Get();  
}
```

Como puedes ver, esta interfaz es muy sencilla, consiste sólo en dos métodos. Cada clase que implemente **ICharQ** necesitará implementar ambos métodos.

2. Crea un archivo llamado **IQDemo.cs**.

3. Comienza a crear **IQDemo.cs** añadiendo la clase **ColaSimple** que se muestra a continuación:

```
// Muestra el funcionamiento de la interfaz ICharQ.  
  
using System;  
  
// Una cola simple, de tamaño fijo, para caracteres.  
class ColaSimple : ICharQ {  
    char[] q; // este arreglo contiene la cola  
    int putloc, getloc; // los índices put y get  
  
    // Construye una cola vacía dado su tamaño.  
    public ColaSimple(int tamaño) {  
        q = new char[tamaño + 1]; // reserva memoria para la cola  
        putloc = getloc = 0;  
    }  
  
    // Coloca un carácter en la cola.  
    public void Put(char ch) {  
        if(putloc==q.Length-1) {  
            Console.WriteLine(" -- La cola está llena.");  
            return;  
        }  
  
        putloc++;  
        q[putloc] = ch;  
    }  
  
    // Obtiene un carácter de la cola.  
    public char Get() {  
        if(getloc == putloc) {  
            Console.WriteLine(" -- La cola está vacía.");  
            return '\0';  
        }  
        return q[getloc++];  
    }  
}
```

```
        return (char)0;
    }

    getloc++;
    return q[getloc];
}
}
```

Esta implementación de **ICharQ** está adaptada de la clase **ColaSimple** mostrada en el capítulo 5 y ya debes estar familiarizado con ella.

- 4.** Añade la clase **ColaCircular** que se muestra a continuación a **IQDemo.cs**. Esto implementa una cola circular para los caracteres.

```
// Una cola circular.
class ColaCircular : ICharQ {
    char[] q; // este arreglo contiene la cola
    int putloc, getloc; // los índices put y get

    // Construye una cola vacía dado su tamaño.
    public ColaCircular(int tamaño) {
        q = new char[tamaño+1]; // reserva memoria para la cola
        putloc = getloc = 0;
    }

    // Coloca un carácter en la cola.
    public void Put(char ch) {
        /* La cola está llena si putloc es uno o menor que
           getloc, o si putloc está al final del arreglo
           y getloc está al principio. */
        if(putloc+1 == getloc ||
           ((putloc==q.Length-1) && (getloc==0))) {
            Console.WriteLine(" -- La cola está llena.");
            return;
        }

        putloc++;
        if(putloc==q.Length) putloc = 0; // loop de regreso
        q[putloc] = ch;
    }

    // Obtiene un carácter de la cola.
    public char Get() {
        if(getloc == putloc) {
            Console.WriteLine(" -- La cola está vacía.");
            return (char) 0;
        }
    }
}
```

```
    getloc++;
    if(getloc==q.Length) getloc = 0; // loop de regreso
    return q[getloc];
}
}
```

La cola circular funciona reutilizando el espacio en el arreglo que queda libre cuando los elementos son recuperados. De esa manera puede almacenar un número ilimitado de elementos, siempre y cuando algunos de ellos sean eliminados. Aunque conceptualmente es sencilla (simplemente restablecer el índice apropiado a cero cuando se llega al final del arreglo), las condiciones del límite son al principio algo confusas. En una cola circular, ésta se considera llena no cuando el arreglo subyacente llega a su fin, sino cuando almacenar otro elemento significaría sobrescribir un elemento que no ha sido recuperado. Así, **Put()** debe verificar varias condiciones para determinar si la cola está llena. Como lo sugieren los comentarios, la cola está llena cuando **putloc** es uno menos que **getloc**, o bien si **putloc** está al final del arreglo cuando **getloc** está al inicio. Como en el caso anterior, la cola está vacía cuando **getloc** y **putloc** son iguales.

5. Finalmente, inserta en **IQDemo.cs** la clase **ColaDin** que se muestra a continuación. Esta clase implementa una cola dinámica; es una versión “incrementable” de la cola circular que expande su tamaño cuando el espacio se agota.

```
// Una cola circular dinámica.
// Esta implementación duplica en automático el
// tamaño de la cola cuando está llena.
class ColaDin : ICharQ {
    char[] q; // este arreglo contiene la cola
    int putloc, getloc; // los índices put y get

    // Construye una cola vacía dado su tamaño.
    public ColaDin(int tamaño) {
        q = new char[tamaño+1]; // reserva memoria para la cola
        putloc = getloc = 0;
    }

    // Coloca un carácter en la cola.
    public void Put(char ch) {
        /* Si la cola está llena, duplica el espacio del
        arreglo subyacente. */
        if(putloc+1 == getloc ||
           ((putloc==q.Length-1) && (getloc==0))) {

            // Incrementa el tamaño del arreglo para la cola.
            char[] t = new char[q.Length * 2];

            // Copia elementos al nuevo arreglo.
            int i;
            for(i=1; putloc != getloc; i++)
                t[i] = Get();
        }
        q[putloc] = ch;
        putloc++;
    }

    // Devuelve el siguiente carácter de la cola.
    public char Get() {
        if(getloc > q.Length)
            throw new IndexOutOfRangeException("Cola vacía");
        return q[getloc];
    }

    // Vacía la cola.
    public void Clear() {
        putloc = getloc = 0;
    }

    // Devuelve el tamaño actual de la cola.
    public int Length {
        get { return q.Length - 1; }
    }
}
```

```
// Restablece los índices getloc y putloc.  
getloc = 0;  
putloc = i-1;  
  
// Hace de q una referencia hacia la nueva cola.  
q = t;  
}  
  
putloc++;  
if(putloc==q.Length) putloc = 0; // loop de regreso  
q[putloc] = ch;  
}  
  
// Obtiene un carácter de la cola.  
public char Get() {  
    if(getloc == putloc) {  
        Console.WriteLine(" -- La cola está vacía.");  
        return (char) 0;  
    }  
  
    getloc++;  
    if(getloc==q.Length) getloc = 0; // loop de regreso  
    return q[getloc];  
}  
}
```

En esta implementación, cuando la cola está llena, intentar almacenar otro elemento provoca que se genere un nuevo arreglo cuyo tamaño es el doble del original; el contenido actual de la cola se copia al nuevo arreglo; se restablecen los índices **putloc** y **getloc**, y una nueva referencia al arreglo se almacena en **q**.

- 6.** Para probar las tres implementaciones de **ICharQ**, inserta la siguiente clase en **IQDemo.cs**. Utiliza una referencia **ICharQ** para accesar las tres colas.

```
// Muestra el funcionamiento de las colas.  
class IQDemo {  
    static void Main() {  
        ColaSencilla q1 = new ColaSencilla(10);  
        ColaDin q2 = new ColaDin(5);  
        ColaCircular q3 = new ColaCircular(10);  
  
        ICharQ IQ;  
  
        char ch;  
        int i;  
  
        // Asigna iq una referencia a una cola simple y fija.  
        iq = q1;
```

(continúa)

```
// Coloca algunos caracteres en la cola.
for(i=0; i < 10; i++)
    iQ.Put((char)('A' + i));

// Muestra el contenido de la cola.
Console.Write("Contenido de la cola de tamaño fijo: ");
for(i=0; i < 10; i++) {
    ch = iQ.Get();
    Console.Write(ch);
}
Console.WriteLine();

// Asigna iQ una referencia a una cola dinámica.
iQ = q2;

// Coloca algunos caracteres en la cola.
for(i=0; i < 10; i++)
    iQ.Put((char)('Z' - i));

// Muestra el contenido de la cola.
Console.Write("Contenido de la cola dinámica: ");
for(i=0; i < 10; i++) {
    ch = iQ.Get();
    Console.Write(ch);
}

Console.WriteLine();

// Asigna iQ una referencia a una cola circular.
iQ = q3;

// Coloca algunos caracteres en la cola circular.
for(i=0; i < 10; i++)
    iQ.Put((char)('A' + i));

// Muestra el contenido de la cola.
Console.Write("Contenido de la cola circular: ");
for(i=0; i < 10; i++) {
    ch = iQ.Get();
    Console.Write(ch);
}

Console.WriteLine();

// Coloca más caracteres en la cola circular.
for(i=10; i < 20; i++)
    iQ.Put((char)('A' + i));
```

```

// Muestra el contenido de la cola.
Console.WriteLine("Contenido de la cola circular: ");
for(i=0; i < 10; i++) {
    ch = iQ.Get();
    Console.Write(ch);
}

Console.WriteLine("\n Almacenamiento y consumo de" +
    " la cola circular.");

// Uso y consumo de la cola circular.
for(i=0; i < 20; i++) {
    iQ.Put((char)('A' + i));
    ch = iQ.Get();
    Console.Write(ch);
}
}
}

```

**7.** Compila el programa incluyendo **ICharQ.cs** e **IQDemo.cs**.

**8.** Los datos de salida generados por el programa son los siguientes:

```

Contenido de la cola de tamaño fijo: ABCDEFGHIJ
Contenido de la cola dinámica: ZYXWVUTSRQ
Contenido de la cola circular: ABCDEFGHIJ
Contenido de la cola circular: KLMNOPQRST
Almacenamiento y consumo de la cola circular.
ABCDEFGHIJKLMNPQRST

```

**9.** He aquí dos cosas que puedes probar por ti mismo. Añade un método **Reset()** a **ICharQ** que restablezca la cola. Crea un método **static** que copie el contenido de un tipo de cola a otro.

## Interfaz para propiedades

Como los métodos, las propiedades se especifican en una interfaz sin cuerpo. He aquí el formato general para una especificación de propiedad:

```

// interfaz de propiedad
nombre tipo {
    get;
    set;
}

```

Por supuesto, sólo **get** y **set** estarán presentes para las propiedades sólo-lectura y sólo-escritura, respectivamente. Aunque declarar las propiedades en una interfaz se asemeja a la declaración de una propiedad autoimplementada en una clase, no son lo mismo. La declaración de interface no provoca que la propiedad se autoimplemente. Sólo especifica el nombre y tipo de la misma. La implementación es delegada a cada clase. De la misma manera, no está permitido el uso de modificadores de acceso cuando la propiedad se declara como **interface**. Así, el accesador **set**, por ejemplo, no puede ser especificado como **private** en una **interface**.

A continuación presentamos una versión modificada de la interfaz **ISeries** y de la clase **Más-Dos** que utiliza una propiedad para obtener y establecer el siguiente elemento en la serie:

```
// Utiliza una propiedad en una interfaz.  
using System;  
  
public interface ISeries {  
    // Una interfaz de propiedad.  
    int Next{ ←———— Declara una propiedad  
        get; // regresa el siguiente número de la serie  
        set; // establece el siguiente número  
    }  
}  
  
// Implementa ISeries.  
class MásDos : ISeries {  
    int val;  
  
    public MásDos() {  
        val = 0;  
    }  
  
    // Obtiene o establece valor.  
    public int Next { ←———— Implementa la propiedad.  
        get {  
            val += 2;  
            return val;  
        }  
        set {  
            val = value;  
        }  
    }  
}  
  
// Muestra la propiedad de interface en acción.  
class ISeriesDemo3 {  
    static void Main() {  
        MásDos ob = new MásDos();  
  
        // Accesa la serie a través de una propiedad.  
        for(int i=0; i < 5; i++)  
            Console.WriteLine("El siguiente valor es " + ob.Next);  
    }  
}
```

```

Console.WriteLine("\nComienza en 21");
ob.Next = 21;
for(int i=0; i < 5; i++)
    Console.WriteLine("El siguiente valor es " + ob.Next);
}
}

```

Los datos generados por el programa son:

```

El siguiente valor es 2
El siguiente valor es 4
El siguiente valor es 6
El siguiente valor es 8
El siguiente valor es 10

```

```

Comienza en 21
El siguiente valor es 23
El siguiente valor es 25
El siguiente valor es 27
El siguiente valor es 29
El siguiente valor es 31

```

## Interfaz para indexadores

Un indexador declarado en una interfaz tiene el siguiente formato:

```

// interfaz de indexador
tipo-elemento this[int índice] {
    get;
    set;
}

```

Como antes, solamente **get** y **set** estarán presentes para los indexadores sólo-lectura y sólo-escritura, respectivamente. No se permite el uso de modificadores de acceso cuando el indexador se declara en una **interface**.

Aquí presentamos otra versión de **ISeries** que añade un indexador de sólo-lectura que regresa el elemento de la posición *i* en la serie:

```

// Añade un indexador en una interfaz.
using System;

public interface ISeries {
    // Una interfaz de propiedad.
    int Next {
        get; // regresa el siguiente número de la serie
        set; // establece el siguiente número
    }
}

```

```
// Una interfaz de indexador.
int this[int index] { ← Declara un indexador sólo-lectura en ISeries.
    get; // regrese el número de la serie especificado
}

// Implementa ISeries.
class MásDos : ISeries {
    int val;

    public MásDos() {
        val = 0;
    }

    // Obtiene o establece un valor utilizando una propiedad.
    public int Next {
        get {
            val += 2;
            return val;
        }
        set {
            val = value;
        }
    }
}

// Obtiene un valor utilizando un indexador.
public int this[int index] { ← Implementa el indexador.
    get {
        val = 0;
        for(int i=0; i < index; i++)
            val += 2;
        return val;
    }
}

// Muestra la interfaz de un indexador en acción.
class ISeriesDemo4 {
    static void Main() {
        MásDos ob = new MásDos();

        // Accesa la serie a través de una propiedad.
        for(int i=0; i < 5; i++)
            Console.WriteLine("El siguiente valor es " + ob.Next);

        Console.WriteLine("\nComienza en 21");
        ob.Next = 21;
        for(int i=0; i < 5; i++)
    }
}
```

```
Console.WriteLine("El siguiente valor es " +
    ob.Next);

Console.WriteLine("\nRestablece a 0");
ob.Next = 0;

// Accesa la serie a través de un indexador.
for(int i=0; i < 5; i++)
    Console.WriteLine("El siguiente valor es " + ob[i]);
}
```

Los datos de salida generados por el programa son:

```
El siguiente valor es 2
El siguiente valor es 4
El siguiente valor es 6
El siguiente valor es 8
El siguiente valor es 10
```

```
Comienza en 21
El siguiente valor es 23
El siguiente valor es 25
El siguiente valor es 27
El siguiente valor es 29
El siguiente valor es 31
```

```
Restablece a 0
El siguiente valor es 0
El siguiente valor es 2
El siguiente valor es 4
El siguiente valor es 6
El siguiente valor es 8
```

## Las interfaces pueden heredarse

Una interfaz puede heredar a otra. La sintaxis es la misma para la herencia entre clases. Cuando una clase implementa una interfaz que hereda otra interfaz, debe implementar todos los miembros definidos dentro de la cadena de herencia. A continuación presentamos un ejemplo:

```
// Una interfaz puede heredar otra interfaz.
using System;

public interface IA {
    void Met1();
    void Met2();
}
```

```
// Ahora IB incluye Met1() y Met2() -- y añade Met3().  
public interface IB : IA { ← IB hereda IA.  
    void Met3();  
}  
  
// Esta clase debe implementar todos los miembros de IA e IB.  
class MiClase : IB {  
    public void Met1() {  
        Console.WriteLine("Implementa Met1().");  
    }  
  
    public void Met2() {  
        Console.WriteLine("Implementa Met2().");  
    }  
  
    public void Met3() {  
        Console.WriteLine("Implementa Met3().");  
    }  
}  
  
class IFExtend {  
    static void Main() {  
        MiClase ob = new MiClase();  
  
        ob.Met1();  
        ob.Met2();  
        ob.Met3();  
    }  
}
```

Como experimento, puedes intentar eliminar la implementación de **Met1()** en **MiClase**. Esto provocará un error en tiempo de compilación. Como se mencionó anteriormente, cualquier clase que implemente una interfaz, debe implementar todos los métodos definidos por esta última, incluyendo los que herede de otras interfaces.

### Pregunta al experto

**P:** Cuando una interfaz hereda otra, ¿es posible declarar un miembro en la interfaz derivada que oculte un miembro definido por la interfaz base?

**R:** Sí. Cuando un miembro en una interfaz derivada tiene la misma firma que otra en la interfaz base, el nombre en la interfaz base se oculta. Como en el caso de las clases heredadas, este ocultamiento provocará un mensaje de advertencia, a menos que especifiques el miembro de la interfaz derivada con la palabra clave **new**.

## Implementaciones explícitas

Cuando se implementa un miembro de interfaz, es posible calificar completamente su nombre con el nombre de la interfaz. Al hacerlo, se crea una *implementación explícita de miembro de interfaz*, o *implementación explícita*, para abbreviar. Por ejemplo, dado

```
interface IMiIF {
    int MiMet(int x);
}
```

es legal implementar **IMiIF** como se muestra a continuación:

```
class MiClase : IMiIF {
    int IMiIF.MiMet(int x) { ← Una calificación completa de nombre utilizado
        return x / 3;
    }
}
```

Como puedes ver, cuando el miembro **MiMet()** de **IMiIF** se implementa, se especifica su nombre completo incluyendo su interfaz.

Existen dos razones por las que necesites crear una implementación explícita de un miembro de interfaz. Primera, es posible que una clase implemente dos interfaces en las cuales se declaren métodos con el mismo nombre y la misma firma de tipo. Al calificar los nombres con sus respectivas interfaces elimina la ambigüedad de esta situación. Segunda, cuando implementas un método con su nombre calificado completamente, estás proporcionando una implementación que *no puede* ser accesada a través de un objeto de la clase. De esta manera, una implementación explícita te ofrece un medio para poner en funcionamiento un método de interfaz que no sea miembro público de la clase que lo implementa. Veamos un ejemplo de cada una.

El siguiente programa contiene una interfaz llamada **IPar**, que define dos métodos, **EsPar()** y **EsNon()**. Estos métodos determinan si un número es par o es non. Luego, **MiClase** implementa **IPar**. Cuando lo hace, implementa **EsNon()** de manera explícita.

```
// Implementa explícitamente un miembro de interfaz.
using System;

interface IPar {
    bool EsNon(int x);
    bool EsPar(int x);
}

class MiClase : IPar {
    // Implementación explícita.
    bool IPar.EsNon(int x) { ← Implementación explícita de EsNon().
        if((x%2) != 0) return true;
        else return false;
    }
}
```

```
// Implementación normal.  
public bool EsPar(int x) {  
    IPar o = this; // Interface referencia al objeto invocado  
  
    return !o.EsNon(x);  
}  
}  
  
class Demo {  
    static void Main() {  
        MiClase ob = new MiClase();  
        bool result;  
  
        result = ob.EsPar(4);  
        if(result) Console.WriteLine("4 es par.");  
        else Console.WriteLine("3 es non.");  
  
        // result = ob.EsNon(); // Error, no expuesto.  
    }  
}
```

Como **EsNon()** se implementa de manera explícita, no se expone como miembro público de **MiClase**. En lugar de ello, **EsNon()** sólo puede ser accesado a través de una referencia de interfaz. Ésta es la razón por la que se invoca a través de **o** en la implementación de **EsPar()**.

He aquí un ejemplo donde se implementan dos interfaces y ambas declaran un método llamado **Met()**. Se utiliza la implementación explícita para eliminar la ambigüedad inherente a la situación.

```
// Utiliza implementación explícita para eliminar ambigüedad.  
using System;  
  
interface IMiIF_A {  
    int Met(int x); ←  
}  
interface IMiIF_B {  
    int Met(int x); ←  
}  
  
// MiClase implementa ambas interfaces.  
class MiClase : IMiIF_A, IMiIF_B {  
    IMiIF_A a_ob;  
    IMiIF_B b_ob;  
  
    // Implementa explícitamente los dos Met().  
    int IMiIF_A.Met(int x) { ←  
        return x + x;  
    }  
}
```

Las firmas de estos dos métodos son las mismas.

La implementación explícita elimina las ambigüedades.

```

int IMiIF_B.Met(int x) { ← La implementación explícita elimina las ambigüedades.
    return x * x;
}

// Invoca Met() a través de una referencia de interfaz.
public int MetA(int x) {
    a_ob = this;
    return a_ob.Met(x); // invoca IMiIF_A
}

public int MetB(int x) {
    b_ob = this;
    return b_ob.Met(x); // invoca IMiIF_B
}

class FQIFNombres {
    static void Main() {
        MiClase ob = new MiClase();

        Console.WriteLine("Invocación de IMiIF_A.Met(): ");
        Console.WriteLine(ob.MetA(3));

        Console.WriteLine("Invocación de IMiIF_B.Met(): ");
        Console.WriteLine(ob.MetB(3));
    }
}

```

Los datos de salida generados por el programa son:

```

Invocación de IMiIF_A.Met(): 6
Invocación de IMiIF_B.Met(): 9

```

Analizando el programa, lo primero que se observa es que **Met()** tiene la misma firma tanto en **IMiIF\_A** como en **IMiIF\_B**. Así, cuando **MiClase** implementa ambas interfaces, lo hace de manera explícita y por separado, calificando completamente su nombre en el proceso. Como la única manera en que puede ser invocado un método implementado explícitamente es por una referencia de interfaz, **MiClase** crea dos de esas referencias, una para **IMiIF\_A** y otra para **IMiIF\_B**. Despues invoca dos de sus propios métodos, los cuales invocan sus respectivos métodos de interfaz, eliminando así la ambigüedad.

## Estructuras

Como sabes, el tipo de las clases es de referencia. Esto significa que los objetos de las clases son accesados a través de una referencia. Esto es diferente de los tipos valor, que son accesados directamente. Sin embargo, puede haber ocasiones en las que pueda ser de utilidad accesar un objeto directamente, del mismo modo como se hace con los tipos valor. Una de las razones de ello es la eficiencia. Accesar objetos de clase a través de referencias añade sobrecarga en cada acceso. También ayuda a conservar espacio. Para cada pequeño objeto, este espacio extra puede ser significan-

te. Para solucionar estos inconvenientes, C# ofrece la estructura. Una *estructura* es similar a una clase, pero es un tipo valor y no un tipo de referencia.

Las estructuras se declaran utilizando la palabra clave **struct** y son sintácticamente similares a las clases. Éste es el formato general de **struct**:

```
struct nombre : interfaces {  
    // declaración de miembros  
}
```

El nombre de la estructura se especifica en *nombre*.

Las estructuras no pueden heredar otras estructuras o clases, ni ser utilizadas como base para otras estructuras o clases. Sin embargo, una estructura puede implementar una o más interfaces. Éstas se especifican después del nombre de la estructura utilizando una lista separada por comas.

Como las clases, los miembros de la estructura incluyen métodos, campos, indexadores, propiedades, métodos de operación y eventos. Las estructuras también pueden definir constructores, pero no destructores. Pero no puedes definir constructores predeterminados (sin parámetros) para una estructura. La razón es que se define automáticamente un constructor predeterminado para todas las estructuras y no puede ser cambiado.

Un objeto de estructura puede ser creado utilizando la palabra clave **new** de la misma manera como se utiliza con un objeto de clase, pero no es indispensable. Cuando se utiliza **new**, se invoca el constructor especificado. Cuando no se utiliza **new**, el objeto se crea pero no es inicializado. En tal caso, necesitarás inicializar manualmente el objeto.

He aquí un ejemplo del uso de la estructura:

```
// Muestra una estructura en acción.  
using System;  
  
// Define una estructura.  
struct Contador { ← Define una estructura.  
    public string nombre;  
    public double balance;  
  
    public Contador(string n, double b) {  
        nombre = n;  
        balance = b;  
    }  
}  
  
// Muestra la estructura Contador.  
class StructDemo {  
    static void Main() {  
        Contador acc1 = new Contador("Tom", 1232.22); // constructor explícito  
        Contador acc2 = new Contador(); // constructor predeterminado  
        Contador acc3; // sin constructor
```

```

Console.WriteLine(acct1.nombre + " tiene un balance de " +
acct1.balance);
Console.WriteLine();

if(acc2.nombre == null) Console.WriteLine("acc2.nombre es null.");
Console.WriteLine("acc2.balance es " + acc2.balance);
Console.WriteLine();

// Se debe inicializar acc3 antes de usarse.
acct3.nombre = "Mary";
acct3.balance = 99.33;
Console.WriteLine(acct3.nombre + " tiene un balance de " +
acct3.balance);
}
}

```

Los datos de salida generados por el programa son:

```

Tom tiene un balance de 1232.22
acc2.nombre es null.
acc2.balance es 0

Mary tiene un balance de 99.33

```

Como lo demuestra el programa, una estructura puede ser inicializada ya sea utilizando **new** para invocar un constructor o simplemente declarando un objeto. Si se utiliza **new**, entonces el campo de la estructura será inicializado, ya sea por el constructor predeterminado que inicializa todos los campos o por un constructor definido por el usuario. Si no se utiliza **new**, entonces el objeto no se inicializa y sus campos deben establecerse antes de poder utilizarlo.

## Pregunta al experto

**P:** Sé que C++ también tiene estructuras y utiliza la palabra clave **struct**. ¿Son lo mismo las estructuras en C# y en C++?

**R:** No. En C++, **struct** define un tipo de clase. Así, en C++, **struct** y **class** son casi equivalentes. (La diferencia tiene que ver con el acceso predeterminado de sus miembros, que es privado para **class** y público para **struct**.) En C#, **struct** define un tipo valor, y **class** un tipo de referencia.

## Enumeraciones

Una *enumeración* es un conjunto de constantes enteras con nombre. Las enumeraciones son comunes en la vida diaria. Por ejemplo, una enumeración de monedas en Estados Unidos sería:

*penny* (centavo), *nickel* (quinto), *dime* (diez centavos), *quarter* (veinticinco centavos), *half dollar* (medio dólar) y *dollar*.

La palabra clave **enum** declara un tipo enumeración. Su formato general es

```
enum nombre { lista enumeración };
```

Aquí el nombre del tipo de la enumeración se especifica con *nombre*. La *lista enumeración* es una lista de identificadores separados por comas. He aquí un ejemplo que define una enumeración llamada **Moneda**:

```
enum Moneda{ Penny, Nickel, Dime, Quarter, HalfDollar, Dollar};
```

El punto clave para comprender una enumeración es que cada símbolo representa un valor entero. Sin embargo, no se define una conversión implícita entre un tipo **enum** y el tipo integral entero, por lo que se debe aplicar una transformación explícita. De la misma manera, se requiere una conversión cuando se hace la transformación entre dos tipos de enumeración. Como las enumeraciones representan valores enteros, puedes utilizar una de ellas para controlar una declaración **switch** o como una variable de control para un loop **for**, por ejemplo.

A cada símbolo de la enumeración se le otorga un valor superior (+1) con relación al símbolo que le antecede. De forma predeterminada, el valor del primer símbolo de enumeración es 0. En consecuencia, en la enumeración **Moneda**, **Penny** es 0, **Nickel** es 1, **Dime** es 2, y así sucesivamente.

Los miembros de una enumeración son accesados por su nombre de tipo a través del operador de punto. Por ejemplo, este código:

```
Console.WriteLine(Moneda.Penny + " " + Moneda.Nickel);
```

muestra en pantalla

```
Penny Nickel
```

He aquí un programa que ilustra la enumeración **Moneda**:

```
// Muestra una enumeración.
using System;

class EnumDemo {
    enum Moneda { Penny, Nickel, Dime, Quarter, HalfDollar, Dollar };

    static void Main() {
        Moneda c; // declara una variable enum

        // Usa c para hacer un ciclo por enum utilizando un loop for.
        for (c = Moneda.Penny; c <= Moneda.Dollar; c++) { ← Una variable
            Console.WriteLine(c + " tiene un valor de " + (int)c); ← Moneda pue-
            // Usa un valor de enumeración para controlar switch. de controlar un
            switch (c) { ← Un valor Moneda puede controlar un switch.
                case Moneda.Nickel:
                    Console.WriteLine("Un nickel son 5 pennies.");
                    break;
                case Moneda.Dime:
                    Console.WriteLine("Un dime son 2 nickels.");
            }
        }
    }
}
```

```
        break;
    case Moneda.Quarter:
        Console.WriteLine("Un quarter son 5 nickels.");
        break;
    case Moneda.HalfDollar:
        Console.WriteLine("Un half-dollar son 5 dimes.");
        break;
    case Moneda.Dollar:
        Console.WriteLine("Un dollar son 10 dimes.");
        break;
    }
    Console.WriteLine();
}
}
}
```

Los datos de salida generados por el programa son:

Penny tiene un valor de 0

Nickel tiene un valor de 1  
Un nickel son 5 pennies.

Dime tiene un valor de 2  
Un dime son 2 nickels.

Quarter tiene un valor de 3  
Un quarter son 5 nickels.

HalfDollar tiene un valor de 4  
Un half-dollar son 5 dimes.

Dollar tiene un valor de 5  
Un dollar son 10 dimes.

Observa que tanto el loop **for** como la declaración **switch** son controladas por **c**, que es una variable de tipo **Moneda**. Como se mencionó, una variable de tipo enumeración puede ser útil para controlar un loop o un **switch**. También observa que la salida de una constante enumeración a través de **WriteLine()** hace que se muestre su nombre. Para obtener su valor, se requiere una transformación **int**.

## Inicializar una enumeración

Puedes especificar el valor de uno o más símbolos de enumeración utilizando un inicializador. Para hacerlo simplemente escribe el signo de igual después de su nombre y a continuación un valor entero. A los símbolos que aparecen después del inicializador se les asigna un valor superior que el valor del inicializador previo. Por ejemplo, el siguiente código asigna el valor de 100 a **Quarter**:

```
enum Moneda { Penny, Nickel, Dime, Quarter=100, HalfDollar, Dollar};
```

Ahora, los valores de estos símbolos son:

|            |     |
|------------|-----|
| Penny      | 0   |
| Nickel     | 1   |
| Dime       | 2   |
| Quarter    | 100 |
| HalfDollar | 101 |
| Dollar     | 102 |

## Especificar el tipo subyacente de una enumeración

De forma predeterminada, las enumeraciones están basadas en tipo **int**, pero puedes crear una de cualquier tipo integral, con excepción de **char**. Para especificar un tipo diferente de **int**, coloca el tipo subyacente después del nombre de la enumeración, separados por dos puntos (:). Por ejemplo, la siguiente declaración hace que la enumeración **Moneda** tenga como base **byte**:

```
enum Moneda : byte { Penny, Nickel, Dime, Quarter=100, HalfDollar, Dollar};
```

Ahora, **Moneda.Penny**, por ejemplo, es una cantidad **byte**.



## Autoexamen Capítulo 9

1. “Una interfaz, múltiples métodos” es un dogma clave de C#, ¿qué característica lo ejemplifica mejor?
2. ¿Cuántas clases puede implementar una interfaz? ¿Cuántas interfaces puede implementar una clase?
3. ¿Se puede heredar una interfaz?
4. ¿Una clase debe implementar todos los miembros de una interfaz?
5. ¿Puede una interfaz declarar un constructor?
6. Crea una interfaz para la clase **Vehículo** del capítulo 8. Llámala **IVehículo**. (Consejo: las propiedades necesitarán soportar operaciones de lectura y escritura, y no pueden tener modificadores de acceso.)
7. Crea una clase para arreglos seguros. Hazlo adaptando el ejemplo final del arreglo falla leve del capítulo 7. (Consejo: la propiedad **Length** debe ser sólo-lectura y no puede ser autoimplementada.)
8. ¿En qué se diferencia una **struct** de una **class**?
9. Muestra cómo crear una enumeración para planetas. Llámala **Planetas**.

# Capítulo 10

## Manejo de excepciones

## Habilidades y conceptos clave

- Fundamentos del manejo de excepciones
  - **try** y **catch**
  - Múltiples cláusulas **catch**
  - Bloques **try** anidados
  - Lanzar una excepción
  - La clase **Exception**
  - **finally**
  - Las excepciones integradas
  - Clases personalizadas de excepción
  - **checked** y **unchecked**
- 

**E**n este capítulo se aborda el manejo de excepciones. Una *excepción* es un error que ocurre durante el tiempo de ejecución. Utilizando el subsistema de manejo de excepciones de C#, puedes manejar errores en tiempo de ejecución de manera estructurada y controlada. La principal ventaja del manejo de excepciones es que automatiza mucho del código de manejo de errores que de otra manera tendría que ser insertado “manualmente” en programas grandes. Por ejemplo, sin el manejo de excepciones un código de error sería devuelto por un método en caso de falla, y este valor tiene que ser verificado manualmente cada vez que el método es invocado. Esta metodología es tediosa y puede provocar más errores. El manejo de excepciones racionaliza el manejo de errores al permitir que tu programa defina un bloque de código llamado *controlador de excepciones*, que se ejecuta de manera automática cuando ocurre un error. No es necesario verificar manualmente el éxito o la falla de cada operación específica o invocación de método. Si ocurre un error, será procesado por el controlador de excepciones.

Otra razón por la que el manejo de errores es importante es que C# define excepciones estándar para errores comunes de programación, como la división entre cero o el desbordamiento de los límites de índice. Para responder a esos errores, tu programa debe buscar y manejar tales excepciones. En el análisis final, ser un programador de C# exitoso significa que eres completamente capaz de navegar por el subsistema de manejo de excepciones del lenguaje.

## La clase System.Exception

En C#, las excepciones están representadas por clases. Todas las clases de excepciones deben derivar de la clase integrada **Exception**, que es parte de la nomenclatura **System**. Así, todas las excepciones son subclases de **Exception**.

Una subclase verdaderamente importante de **Exception** es **SystemException**. Ésta es la clase de la cual se derivan todas las excepciones generadas por el sistema de tiempo de ejecución de C# (es decir, el CLR). **SystemException** no añade nada a **Exception**, simplemente define la parte más alta de la jerarquía estándar de excepciones.

.NET Framework define varias excepciones integradas derivadas de **SystemException**. Por ejemplo, cuando se intenta realizar una división entre cero, se genera una excepción **DivideByZero**. Como verás más adelante en este mismo capítulo, puedes crear tus propias clases de excepción.

## Fundamentos del manejo de excepciones

El manejo de excepciones en C# es administrado por cuatro palabras clave: **try**, **catch**, **throw** y **finally**. Forma un subsistema interrelacionado donde el uso de una implica el uso de otra. A lo largo de este capítulo se examina cada una con detalle. Sin embargo, resulta de utilidad tener desde este momento una comprensión general del papel que tiene cada una en el manejo de excepciones. A continuación, y de manera muy breve, explicaremos cómo funcionan.

Las declaraciones del programa que quieras monitorear en busca de excepciones están contenidas dentro de un bloque **try**. Si ocurre una excepción dentro de ese bloque, la misma pasa al estatus **throw**, es decir, es *lanzada*. Tu código puede atrapar esta excepción utilizando **catch** y manejarla de una manera comprensible. Las excepciones generadas por el sistema son lanzadas en automático por el sistema de tiempo de ejecución. Para lanzar manualmente una excepción, utiliza la palabra clave **throw**. Cualquier código que deba ser ejecutado de manera necesaria, incluso sobre la existencia de un bloque **try**, se coloca en un bloque **finally**.

### Utilizar try y catch

En el corazón del manejo de excepciones se encuentran **try** y **catch**. Estas palabras clave funcionan juntas, y no puedes dejar un **catch** sin un **try**. He aquí el formato general de los bloques para manejo de excepciones **try/catch**:

```
try {  
    // bloque de código para monitorear en busca de errores  
}  
  
catch (ExcepTipo1 exOb) {  
    // manejo de ExcepTipo1  
}  
  
catch (ExcepTipo2 exOb) {  
    // manejo de ExcepTipo2  
}  
.  
.  
.
```

Aquí, *ExcepTipo* es el tipo de excepción que está ocurriendo. Cuando se lanza una excepción, ésta es capturada por su correspondiente cláusula **catch**, la cual luego procesa la excepción. Como lo muestra el formato general, puede haber más de una cláusula **catch** asociada con **try**. El tipo de

excepción determina cuál **catch** se ejecutará. Es decir, si el tipo de excepción especificado por **catch** coincide con el de la excepción, entonces el bloque de código asociado con esa cláusula **catch** será ejecutado (y todas las demás cláusulas **catch** serán ignoradas). Cuando se atrapa una excepción, la variable de excepción *exOb* recibirá su valor.

De hecho, es opcional especificar *exOb*. Si el controlador de excepciones no necesita acceso al objeto excepción (como es el caso regularmente), entonces no es necesario especificar *exOb*. Por esta razón, muchos ejemplos en este capítulo no harán esa especificación.

He aquí un punto importante: si no se lanza ninguna excepción, entonces el bloque **try** termina de manera normal, y todas las cláusulas **catch** son ignoradas. La ejecución del programa continúa con la primera declaración inmediata posterior al último **catch**. Así, **catch** se ejecuta únicamente si se lanza una excepción.

## Un sencillo ejemplo de excepción

A continuación presentamos un sencillo ejemplo que ilustra cómo esperar y atrapar una excepción. Como sabes, es un error intentar indexar un arreglo más allá de sus límites. Cuando esto ocurre, el sistema en tiempo de ejecución de C# lanza una excepción estándar **IndexOutOfRangeException**. El siguiente programa genera de modo intencional una excepción de ese tipo y la atrapa:

```
// Muestra el control de excepciones.
using System;

class ExcDemo1 {
    static void Main() {
        int[] nums = new int[4];

        try {
            Console.WriteLine("Antes de que se genere la excepción.");

            // Genera una excepción de índice fuera de límites.
            nums[7] = 10; ← Intenta insertar un índice fuera
                           de los límites de nums.
            Console.WriteLine("esto no se mostrará");
        }
        catch (IndexOutOfRangeException) { ← Atrapa la excepción.
            // atrapa la excepción
            Console.WriteLine("índice fuera de límites!");
        }
        Console.WriteLine("Después del bloque catch.");
    }
}
```

El programa muestra los siguientes datos de salida:

```
Antes de que se genere la excepción.
índice fuera de límites!
Después del bloque catch.
```

Aunque muy breve, el anterior programa ilustra varios puntos clave sobre el control de excepciones. Primero, el código que quieras monitorear por errores está contenido dentro del bloque **try**. Segundo, cuando ocurre una excepción (en este caso el intento de indexar **nums** más allá de su límite), la excepción es lanzada por el bloque **try** y es capturada por **catch**. En este punto, el control

pasa al bloque **catch**, y termina el bloque **try**. Es decir, **catch** no es invocado. En lugar de eso, la ejecución del programa se le transfiere. Así, la declaración **WriteLine()** que sigue al índice fuera de límite nunca se ejecutará. Después de que se ejecuta el bloque **catch**, el control del programa continúa con las declaraciones posteriores a **catch**. De esta manera, es obligación de tu controlador de excepciones remediar el problema que provoca la excepción de modo que la ejecución del programa continúe con normalidad.

Observa que no se especifica un nombre de variable en la cláusula **catch**. En lugar de eso, sólo se requiere el tipo de excepción (en este caso, **IndexOutOfRangeException**). Como se mencionó, una variable de excepción sólo se necesita cuando el acceso a objetos de excepción es requerido. En algunos casos, el valor del objeto de excepción puede ser utilizado por el controlador para obtener información adicional sobre el error, pero en muchos casos, basta con saber que una excepción ha ocurrido. Por lo mismo, no es raro que la variable de excepción esté ausente, como en el programa anterior.

Como se explicó, si el bloque **try** no lanza ninguna excepción, no se ejecutarán ningún **catch** y el control del programa se reasume después de **catch**. Para confirmarlo, en el programa anterior cambia la línea

```
nums[ 7 ] = 10;
```

por

```
nums[ 0 ] = 10;
```

En este caso no se genera ninguna excepción y el bloque **catch** no se ejecuta.

## Un segundo ejemplo de excepción

Es importante comprender que todo el código dentro del bloque **try** se monitorea en busca de excepciones. Éstas incluyen las que puedan ser generadas por la invocación de un método dentro del bloque **try**. Una excepción lanzada por la invocación de un método dentro del bloque **try** puede ser capturada ahí mismo, suponiendo que, por supuesto, el método en sí no captura la excepción. Por ejemplo, el siguiente es un programa válido:

```
// Una excepción puede ser generada por un método y capturada por otro.
using System;

class ExcTest {
    // Genera una excepción.
    public static void GenException() {
        int[] nums = new int[4];

        Console.WriteLine("Antes de que se genere la excepción.");

        // Genera una excepción de índice fuera de límite.
        nums[7] = 10; ← La excepción se genera aquí.
        Console.WriteLine("esto no será mostrado");
    }
}
```

```
class ExcDemo2 {
    static void Main() {
        try {
            ExcTest.GenException();
        }
        catch (IndexOutOfRangeException) { ← La excepción se captura aquí.
            // Atrapa la excepción.
            Console.WriteLine("¡Índice fuera de límites!");
        }
        Console.WriteLine("Después del bloque catch.");
    }
}
```

El programa genera los siguientes datos de salida, que son los mismos que los producidos por la primera versión mostrada con anterioridad:

```
Antes de que se genere la excepción.
¡Índice fuera de límites!
Después del bloque catch.
```

Como se invoca **GenException()** dentro del bloque **try**, la excepción que genera (pero no captura) es atrapada por el **catch** en **Main()**. Sin embargo, debes comprender que si **GenException()** hubiera atrapado la excepción, ésta nunca hubiese pasado de regreso a **Main()**.

## Las consecuencias de una excepción sin atrapar

Capturar una de las excepciones estándar de C#, como lo hace el programa anterior, tiene un beneficio secundario: evita que el programa termine de manera anormal. Cuando se lanza una excepción, ésta debe ser atrapada por algún segmento de código en alguna parte. En general, si tu programa no captura una excepción, entonces será capturada por el sistema de tiempo de ejecución. El problema es que este sistema reportará un error y terminará el programa de manera abrupta. Por ejemplo, en la siguiente versión del programa anterior, la excepción del índice fuera de límites no es capturada por el programa:

```
// Deja que el sistema de tiempo de ejecución controle el error.
using System;

class SinControlador {
    static void Main() {
        int[] nums = new int[4];

        Console.WriteLine("Antes de que se genere la excepción.");

        // Genera una excepción de índice fuera de límite.
        nums[7] = 10;
    }
}
```

Cuando ocurre el error en el índice del arreglo, la ejecución del programa se detiene y se muestra el siguiente mensaje de error:

Antes de que se genere la excepción:

```
Unhandled Exception: System.IndexOutOfRangeException:  
    Index was outside the bounds of the array.  
  at SinControlador.Main()
```

(Excepción sin controlar: System.IndexOutOfRangeException: El índice estaba fuera de los límites del arreglo. en SinControlador.Main( ).)

Este mensaje es de utilidad para el programador durante la depuración, pero no es algo que quieras que el usuario de tu programa vea, ¡por decir lo menos! Por ello es importante que tu programa controle las excepciones por sí mismo.

Como se mencionó anteriormente, el tipo de excepción debe coincidir con el especificado en la cláusula **catch**. De no ser así, la excepción no será capturada. Por ejemplo, el siguiente programa intenta capturar un error de límites de arreglo con un **catch** que controla la excepción **DivideByZeroException** (otra excepción integrada). Cuando se desborda el límite del arreglo, se genera la excepción **IndexOutOfRangeException**, pero no será atrapada por **catch**. Esto trae como resultado una terminación anormal del programa.

```
// Esto no funcionará.  
using System;  
  
class ExcTipoDiscordante {  
    static void Main() {  
        int[] nums = new int[4];  
  
        try {  
            Console.WriteLine("Antes de que se genere la excepción.");  
  
            // Genera una excepción de índice fuera de límite.  
            nums[7] = 10; ←  
            Console.WriteLine("esto no se mostrará"); ← Esto lanza una excepción  
        }   IndexOutOfRangeException.  
  
        /* No es posible atrapar un error de límites  
         * con una excepción DivideByZeroException. */  
        catch (DivideByZeroException) { ← Pero esto intenta capturar-  
            Console.WriteLine("Índice fuera de límites!"); ← la con una excepción  
        }   DivideByZeroException.  
        Console.WriteLine("Después del bloque catch.");  
    }  
}
```

Los datos de salida generados por el programa son:

Antes de que se genere la excepción.

```
Unhandled Exception: System.IndexOutOfRangeException:  
    Index was outside the bounds of the array.  
  at ExcTipoDiscordante.Main()
```

(Excepción sin controlar: System.IndexOutOfRangeException: El índice estaba fuera de los límites del arreglo. en ExcTipoDiscordante.Main( ).)

Como lo demuestran los datos de salida, un **catch** para **DivideByZeroException** no atrapará una **IndexOutOfRangeException**.

## Las excepciones te permiten controlar errores con elegancia

Uno de los beneficios clave del control de excepciones es que permite a tu programa responder a un error y continuar con la ejecución. Por ejemplo, analiza el siguiente programa que divide los elementos de un arreglo entre los elementos de otro. Si ocurre una división entre cero, se genera una **DivideByZeroException**. En el programa, esta excepción es manejada reportando el error y continuando con la ejecución. Así, el intento de dividir entre cero no termina de forma abrupta el programa por generar un error en tiempo de ejecución. En lugar de ello, se maneja con elegancia, permitiendo que el programa continúe con su ejecución.

```
// Manejar con elegancia los errores y continuar.
using System;

class ExcDemo3 {
    static void Main() {
        int[] numer = { 4, 8, 16, 32, 64, 128 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i < numer.Length; i++) {
            try {
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " es " +
                    numer[i]/denom[i]);
            }
            catch (DivideByZeroException) {
                Console.WriteLine("¡No es posible dividir entre cero!");
            }
        }
    }
}
```

Los datos generados por el programa son:

```
4 / 2 es 2
¡No es posible dividir entre cero!
16 / 4 es 4
32 / 4 es 8
¡No es posible dividir entre cero!
128 / 8 es 16
```

Este ejemplo tiene otro punto de importancia: una vez que se ha controlado una excepción, se elimina del sistema. Por tanto, en el ejemplo, cada pase a través del loop inserta un bloque **try** nuevo; todas las excepciones anteriores se han controlado. Esto permite que el programa controle errores repetitivos.

## Utilizar múltiples cláusulas catch

Puedes asociar más de una cláusula **catch** con un bloque **try**. De hecho, es una práctica común. No obstante, cada **catch** debe atrapar diferentes tipos de excepciones. Por ejemplo, el programa que se muestra a continuación atrapa errores de desbordamiento de arreglos y de división entre cero:

```
// Utiliza múltiples cláusulas catch.
using System;

class ExcDemo4 {
    static void Main() {
        // Aquí, numer es mayor que denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i < numer.Length; i++) {
            try {
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " es " +
                    numer[i] / denom[i]);
            }
            catch (DivideByZeroException) { ← Múltiples cláusulas catch.
                Console.WriteLine("¡No es posible dividir entre cero!");
            }
            catch (IndexOutOfRangeException) { ←
                Console.WriteLine("No se encontraron elementos coincidentes.");
            }
        }
    }
}
```

El programa produce los siguientes datos de salida:

```
4 / 2 es 2
¡No es posible dividir entre cero!
16 / 4 es 4
32 / 4 es 8
¡No es posible dividir entre cero!
128 / 8 es 16
No se encontraron elementos coincidentes.
No se encontraron elementos coincidentes.
```

Como lo confirman los datos de salida, cada cláusula **catch** responde sólo a su propio tipo de excepción.

En general, las cláusulas **catch** se verifican en el orden en que aparecen dentro del programa. Sólo las cláusulas coincidentes se ejecutan; el resto son ignoradas.

## Atrapar todas las excepciones

En algunas ocasiones querrás atrapar todas las excepciones, sin importar el tipo. Para hacerlo, utiliza cláusulas **catch** que no especifiquen ningún tipo. Esto crea un controlador “atrapa todo” que es útil cuando quieras asegurarte de que todas las excepciones son controladas en tu programa. Por ejemplo, en el siguiente programa el único **catch** es un “atrapa todo”, y atrapa tanto excepciones **IndexOutOfRangeException** como **DivideByZeroException**; ambas son generadas por el programa:

```
// Utiliza un catch "atrapa todo".
using System;

class ExcDemo5 {
    static void Main() {
        // Aquí, numer es mayor que denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i < numer.Length; i++) {
            try{
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " es " +
                    numer[i] / denom[i]);
            }
            catch { ← Esto atrapa todas las excepciones.
                Console.WriteLine("Ocurrieron algunas excepciones.");
            }
        }
    }
}
```

Los datos generados por el programa son:

```
4 / 2 es 2
Ocurrieron algunas excepciones.
16 / 4 es 4
32 / 4 es 8
Ocurrieron algunas excepciones.
128 / 8 es 16
Ocurrieron algunas excepciones.
Ocurrieron algunas excepciones.
```

## Los bloques try se pueden anidar

Un bloque **try** puede anidarse dentro de otro. Una excepción generada dentro del bloque interno que no es capturada por el **catch** asociado con ese **try** se propaga al bloque **try** externo.

Por ejemplo, en este caso **IndexOutOfRangeException** no es atrapada por el bloque **try** interno, sino por el **try** externo:

```
// Uso de bloques try anidados.
using System;

class AnidaTrys {
    static void Main() {
        // Aquí, numer es mayor que denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        try { // try externo ←
            for(int i=0; i < numer.Length; i++) {
                try{ // try anidado ←
                    Console.WriteLine(numer[i] + " / " +
                        denom[i] + " es " +
                        numer[i] / denom[i]);
                }
                catch (DivideByZeroException) { // atrapado por el try interno
                    Console.WriteLine("¡No es posible dividir entre cero!");
                }
            }
            catch (IndexOutOfRangeException) { // atrapado por el try externo
                Console.WriteLine("No se encontraron elementos coincidentes.");
                Console.WriteLine("Error fatal -- terminación del programa.");
            }
        }
    }
}
```

Bloques **try** anidados.

Los datos de salida generados por el programa son:

```
4 / 2 es 2
¡No es posible dividir entre cero!
16 / 4 es 4
32 / 4 es 8
¡No es posible dividir entre cero!
128 / 8 es 16
No se encontraron elementos coincidentes.
Error fatal -- terminación del programa.
```

En el ejemplo, una excepción que puede ser controlada por el **try** interno (en este caso el error de dividir entre cero) permite que el programa continúe. Sin embargo, un error por desbordamiento de arreglo es capturado por el **try** externo, lo que provoca la terminación del programa.

Aunque de ninguna manera es la única razón para anidar bloques **try**, el programa anterior presenta un punto de importancia que puede generalizarse. Muchas veces, los bloques **try** anida-

dos se utilizan para permitir que diferentes categorías de errores sean controlados de diversas maneras. Algunos tipos de errores son catastróficos y no pueden arreglarse. Algunos son menores y pueden controlarse de inmediato. Muchos programadores utilizan un bloque **try** externo para capturar los errores más severos, dejando que los bloques **try** internos se arreglen con los problemas menos serios. También puedes utilizar un bloque **try** externo como “atrapa todo” para los errores que no son capturados por los bloques internos.

## Lanzar una excepción

Los ejemplos anteriores han atrapado excepciones generadas de manera automática por el sistema en tiempo de ejecución. No obstante, es posible lanzar manualmente una excepción utilizando la declaración **throw**. Su formato general es el siguiente:

```
throw exceptOb;
```

Aquí, *exceptOb* debe ser una instancia de una clase de excepción derivada de **Exception**.

He aquí un ejemplo que ilustra la declaración **throw** lanzando en forma manual **DivideByZeroException**:

```
// Lanza manualmente una excepción.
using System;

class ThrowDemo {
    static void Main() {
        try {
            Console.WriteLine("Antes de throw.");
            throw new DivideByZeroException(); ← Lanza una excepción.
        }
        catch (DivideByZeroException) {
            Console.WriteLine("Excepción capturada.");
        }
        Console.WriteLine("Después de la declaración try/catch.");
    }
}
```

Los datos que genera el programa son:

```
Antes de throw.
Excepción capturada.
Después de la declaración try/catch.
```

Observa que **DivideByZeroException** fue creada utilizando **new** en la declaración **throw**. Recuerda, **throw** lanza un objeto. Por tanto, debes crear un objeto que pueda lanzar; no puedes lanzar un tipo. En este caso, el constructor predeterminado se utiliza para crear un objeto **DivideByZeroException**, pero están disponibles otros constructores para excepciones (como verás más adelante).

## Pregunta al experto

**P:** ¿Por qué querría lanzar manualmente una excepción?

**R:** La mayoría de las veces, las excepciones que lanzarás serán instancias de clases de excepción que tú mismo crees. Como verás más adelante en este mismo capítulo, crear tus propias clases de excepción te permite controlar errores en el programa como parte de tu estrategia global de control de excepciones.

## Volver a lanzar una excepción

Una excepción capturada por una cláusula **catch** puede ser lanzada de nuevo para que la capture un **catch** externo. La razón más convincente para realizar esta tarea es permitir que múltiples controladores tengan acceso a la excepción en cuestión. Por ejemplo, quizás un controlador de excepción maneje un aspecto de la misma y un segundo controlador haga frente a otro aspecto. Para volver a lanzar una excepción, simplemente declaras **throw**, sin especificar una excepción. Es decir, utilizas este formato:

```
throw ;
```

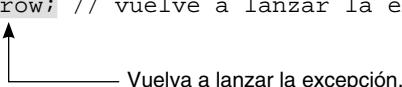
Recuerda que cuando vuelves a lanzar una excepción no será atrapada de nuevo por la misma cláusula **catch**. Se propagará a la cláusula **catch** externa.

El siguiente programa ilustra el relanzamiento de una excepción:

```
// Vuelve a lanzar una excepción.
using System;

class Relanza {
    public static void GenException() {
        // Aquí, numer es mayor que denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i < numer.Length; i++) {
            try {
                Console.WriteLine(numer[i] + " / " +
                    denom[i] + " es " +
                    numer[i] / denom[i]);
            }
            catch (DivideByZeroException) {
                Console.WriteLine("¡No es posible dividir entre cero!");
            }
            catch (IndexOutOfRangeException) {
                Console.WriteLine("No se encontró elemento coincidente.");
                throw; // vuelve a lanzar la excepción
            }
        }
    }
}
```



Vuelva a lanzar la excepción.

```
class RelanzaDemo {
    static void Main() {
        try {
            Relanza.GenException();
        }
        catch (IndexOutOfRangeException) { ← Atrapa la excepción que se volvió a lanzar.
            // vuelve a atrapar excepción
            Console.WriteLine("Error fatal -- programa terminado.");
        }
    }
}
```

En este programa los errores de división entre cero se controlan localmente mediante **GenException()**, pero regresa un error en la colección. En este caso, se detiene con **Main()**.

## Utilizar finally

En ocasiones querrás definir un bloque de código que se ejecute cuando **try/catch** hayan concluido. Por ejemplo, una excepción puede provocar un error que termine el método en ejecución, causando un regreso prematuro. Sin embargo, es posible que ese método haya dejado abierto un archivo en una conexión de red que necesite cerrarse. Tales circunstancias son comunes en programación y C# proporciona un medio conveniente para manejarlas: **finally**.

Para especificar un bloque de código que se ejecute cuando el programa sale de **try/catch**, incluye un bloque **finally** al final de la secuencia **try/catch**. El formato general de **try/catch** que incluye **finally** se muestra a continuación:

```
try {
    // bloque de código para monitorear errores
}

catch (ExcepTipo1 exOb) {
    // control para ExcepTipo1
}

catch (ExcepTipo2 exOb) {
    // control para ExcepTipo2
}

// ...

finally {
    // código final
}
```

El bloque **finally** se ejecutará cuando la ejecución del programa abandone el bloque **try/catch**, sin importar las condiciones que lo hayan provocado. Es decir, ya sea que el bloque **try** termi-

ne de manera normal o a causa de una excepción, el último código que se ejecutará será el definido por **finally**. Este bloque también se ejecuta si cualquier código dentro del bloque **try** o cualesquiera de sus cláusulas **catch** regresan del método.

He aquí un ejemplo de **finally**:

```
// Uso de finally.
using System;

class UsaFinally {
    public static void GenExcep(int w) {
        int t;
        int[] nums = new int[2];

        Console.WriteLine("Recibe " + w);
        try {
            switch (w) {
                case 0:
                    t = 10 / w; // genera error de división entre cero.
                    break;
                case 1:
                    nums[4] = 4; // genera error de índice en arreglo.
                    break;
                case 2:
                    return; // regresa del bloque try
            }
        }
        catch (DivideByZeroException) {
            Console.WriteLine("¡No es posible dividir entre cero!");
            return; // regresa de catch
        }
        catch (IndexOutOfRangeException) {
            Console.WriteLine("No se encontraron elementos coincidentes.");
        }
        finally { ←
            Console.WriteLine("Abandona try.");
        }
    }

    class FinallyDemo {
        static void Main() {

            for(int i=0; i < 3; i++) {
                UsaFinally.GenExcep(i);
                Console.WriteLine();
            }
        }
    }
}
```

El bloque **finally** se ejecuta cuando se abandonan los bloques **try/catch**.

Los datos de salida generados por el programa son:

```
Recibe 0
¡No es posible dividir entre cero!
Abandona try.

Recibe 1
No se encontraron elementos coincidentes.
Abandona try.

Recibe 2
Abandona try.
```

Como lo muestran los datos de salida, no importa cómo termine el bloque **try**, el bloque **finally** se ejecuta de cualquier modo.

Otro punto: Desde la perspectiva de la sintaxis, cuando un bloque **finally** sigue a uno **try**, técnicamente no se requieren cláusulas **catch**. De esta manera, puedes tener un bloque **try** seguido por un **finally** sin cláusulas **catch** intermedias. En tal caso, el bloque **finally** se ejecuta cuando existe **try**, pero no se controla ninguna excepción.

## Análisis detallado de las excepciones

Hasta este punto hemos atrapado excepciones, pero no hemos hecho nada con el objeto de excepción en sí. Como se explicó antes, una cláusula **catch** te permite especificar un tipo de excepción y una variable. La variable recibe una referencia al objeto excepción. Como todas las excepciones son derivadas de la clase **Exception**, todas las excepciones soportan los miembros definidos por esa clase. Aquí examinaremos varios de sus miembros y constructores de mayor utilidad, y pondremos en funcionamiento la variable de excepción.

**Exception** define varias propiedades. Tres de las más interesantes son: **Message**, **StackTrace** y **TargetSite**. Todas ellas son sólo-lectura. **Message** es una cadena de caracteres que describe la naturaleza del error. **StackTrace** es una cadena de caracteres que contiene la pila de invocaciones que condujeron a la excepción. **TargetSite** regresa un objeto que especifica el método que generó la excepción.

**Exception** también define varios métodos. El que utilizarás con mayor frecuencia será **ToString()**, que regresa una cadena de caracteres que describe la excepción. **ToString()** se invoca automáticamente cuando una excepción se muestra a través de **WriteLine()**, por ejemplo.

El siguiente programa muestra las propiedades y los métodos que acabamos de mencionar.

```
// Utiliza miembros de excepción.
using System;

class ExcTest {
    public static void GenExcep() {
        int[] nums = new int[4];

        Console.WriteLine("Antes de que se genere la excepción.");
        // Genera una excepción de desbordamiento de límites en un índice.
        nums[7] = 10;
        Console.WriteLine("esto no se mostrará");
    }
}
```

```

class UsaExcept {
    static void Main() {

        try {
            ExcTest.GenExcep();
        }
        catch (IndexOutOfRangeException exc) {
            Console.WriteLine("El mensaje estándar es: ");
            Console.WriteLine(exc); // invoca ToString()
            Console.WriteLine("StackTrace: " + exc.StackTrace);
            Console.WriteLine("Message: " + exc.Message);
            Console.WriteLine("TargetSite: " + exc.TargetSite);
        }
        Console.WriteLine("Después del bloque catch.");
    }
}

```

Los datos de salida generados por el programa son los siguientes:

```

Antes de que se genere la excepción.
El mensaje estándar es:
System.IndexOutOfRangeException: Index was outside the bounds of the
array.
(Índice fuera de los límites del arreglo)

at ExcTest.GenExcep()
at UsaExcept.Main()
StackTrace: at ExcTest.GenExcep()
at UsaExcept.Main()
Message: Index was outside the bounds of the array.
(Índice fuera de los límites del arreglo)
TargetSite: Void GenExcep()
Después del bloque catch.

```

**Exception** define los siguientes cuatro constructores:

```

public Exception()

public Exception(string str)

public Exception(string str, Exception inner)

protected Exception(System.Runtime.Serialization.SerializationInfo si,
                   System.Runtime.Serialization.StreamingContext sc)

```

El primero es el constructor predeterminado. El segundo especifica la cadena de caracteres relacionada con la propiedad **Message** asociada con la excepción. El tercero especifica lo que se llama una *excepción interna*. Se utiliza cuando una excepción hace que surja otra. En este caso, *inner* especifica la primera excepción, la cual será nula en caso de que no exista una excepción interna. (La excepción interna, si existe, puede obtenerse de la propiedad **InnerException** definida por **Exception**.) El último constructor controla excepciones que ocurren remotamente y requieren deserialización. (La ejecución remota y la serialización rebasan los objetivos de este libro.)

| Excepción                  | Significado                                                                    |
|----------------------------|--------------------------------------------------------------------------------|
| ArrayTypeMismatchException | El tipo de valor que se almacena es incompatible con el tipo del arreglo.      |
| DivideByZeroException      | Intento de dividir entre cero.                                                 |
| IndexOutOfRangeException   | El índice del arreglo está fuera del límite.                                   |
| InvalidOperationException  | Una transformación de tipos en tiempo de ejecución es inválida.                |
| OutOfMemoryException       | Una invocación a <b>new</b> falló porque no hay suficiente memoria disponible. |
| OverflowException          | Ocurrió un desbordamiento aritmético.                                          |
| StackOverflowException     | La pila se desbordó.                                                           |

**Tabla 10-1** Excepciones más comunes definidas dentro de la nomenclatura **System**

Otro punto: en los cuatro constructores **Exception** mostrados anteriormente, observa que los tipos **SerializationInfo** y **StreamingContext** son antecedidos por **System.Runtime.Serialization**. Esto especifica la nomenclatura que los contiene. Las nomenclaturas son explicadas con detalle en el capítulo 14.

## Excepciones de uso común

La nomenclatura **System** define varias excepciones integradas estándar. Todas derivadas de **SystemException**, dado que son generadas por el CLR cuando ocurren errores en tiempo de ejecución. En la tabla 10-1 se muestran varias de las excepciones estándar de mayor uso.

## Derivar clases de excepción

Aunque las excepciones integradas de C# controlan los errores más comunes, su mecanismo para el control de excepciones no se limita a esos errores. De hecho, parte del poder de C# respecto a las excepciones es su capacidad para manejar excepciones que tú mismo creas. Puedes utilizar excepciones personalizadas para controlar errores en tu propio código. Crear una excepción es una tarea fácil. Simplemente define una clase derivada de **Exception**. Tus clases derivadas no necesitan implementar nada; es su existencia en el sistema de tipos lo que te permite utilizarlas como excepciones.

### NOTA

En el pasado, las excepciones personalizadas se derivaban de **ApplicationException**, porque era la jerarquía reservada originalmente para excepciones relacionadas con la aplicación. Sin embargo, Microsoft ya no recomienda esta práctica. En lugar de ello, en el momento de escribir este libro, Microsoft recomienda derivar excepciones personalizadas de **Exception**. Por tal razón, ésa será la técnica que utilizaremos.

Las clases de excepción que crees de manera automática tendrán disponibles las propiedades y los métodos definidos por **Exception**. Por supuesto, puedes anular uno o más de estos miembros en las clases de excepción que crees.

Cuando creas tu propia clase de excepción, por lo regular querrás que soporte todos los constructores definidos por **Exception**. Para las clases de excepción personalizadas sencillas, esto es fácil de hacer porque simplemente tienes que transmitir los argumentos del constructor al correspondiente constructor **Exception** a través de **base**. Por supuesto, técnicamente sólo necesitas proporcionar aquellos constructores que está utilizando el programa.

A continuación presentamos un ejemplo que crea una excepción llamada **NonIntResultException**. En el programa, esta excepción es lanzada cuando el resultado de dividir dos valores enteros produce un resultado con un componente decimal. Para efectos de ilustración, **NonIntResultException** define todos los constructores estándar, aunque la mayoría no se utilizan en el ejemplo. También anula el método **ToString()**.

```
// Utiliza excepciones personalizadas.
using System;

// Crea una excepción.
class NonIntResultException : Exception { ← Una excepción personalizada.
    /* Implementa todos los constructores de excepción. Observa que
       los constructores simplemente ejecutan el constructor base.
       Como NonIntResultException no añade nada a Exception,
       no hay necesidad de acciones posteriores. */
    public NonIntResultException() : base() { }
    public NonIntResultException(string str) : base(str) { }
    public NonIntResultException(string str, Exception inner) :
        base(str, inner) { }
    protected NonIntResultException(
        System.Runtime.Serialization.SerializationInfo si,
        System.Runtime.Serialization.StreamingContext sc) :
        base(si, sc) { }

    // Elimina ToString para NonIntResultException.
    public override string ToString() { ← Elimina ToString().
        return Message;
    }
}

class ExceptPersoDemo {
    static void Main() {

        // Aquí, numer contiene algunos valores aleatorios.
        int[] numer = { 4, 8, 15, 32, 64, 127, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };

        for(int i=0; i < numer.Length; i++) {
            try {
                if((numer[i] % denom[i]) != 0)
                    throw new ← Lanza una excepción personalizada.
                        NonIntResultException("Resultado de " +
                        numer[i] + " / " + denom[i] + " no es par.");
            }
        }
    }
}
```

```
        Console.WriteLine(numer[i] + " / " +
                           denom[i] + " es " +
                           numer[i] / denom[i]);
    }
    catch (DivideByZeroException) {
        Console.WriteLine("¡No es posible dividir entre cero!");
    }
    catch (IndexOutOfRangeException) {
        Console.WriteLine("No se encontraron elementos coincidentes.");
    }
    catch (NonIntResultException exc) {
        Console.WriteLine(exc);
    }
}
}
```

Los datos generados por el programa son:

```
4 / 2 es 2
¡No es posible dividir entre cero!
Resultado de 15 / 4 no es par.
32 / 4 es 8
¡No es posible dividir entre cero!
Resultado de 127 / 8 no es par.
No se encontraron elementos coincidentes.
No se encontraron elementos coincidentes.
```

Observa que ninguno de los constructores proporcionan ninguna declaración en su cuerpo. En lugar de ello, simplemente transmiten sus argumentos a **Exception** a través de **base**. En los casos en que tus clases de excepción no añaden funcionalidad, simplemente dejas que los constructores **Exception** controlen el proceso. Como se explicó, no es necesario que tus clases derivadas añadan nada a lo que heredan de **Exception**. Es su existencia en el sistema de tipos lo que te permite utilizarlas como excepciones.

Antes de continuar, quizás quieras experimentar un poco con este programa. Por ejemplo, intenta comentar la anulación de **ToString()** y observa el resultado. De la misma manera, intenta crear una excepción utilizando el constructor establecido y observa lo que C# genera como su mensaje predeterminado.

## Atrapar excepciones de clases derivadas

Debes tener cuidado en la manera en que ordenas las cláusulas **catch** cuando intentas atrapar tipos de excepción que involucran clases base y sus derivadas, porque una cláusula **catch** para una clase base también coincidirá con cualquiera de sus clases derivadas. Por ejemplo, como la clase base para todas las excepciones es **Exception**, al atraparla se atrapan todas las posibles excepciones. Por supuesto, utilizar **catch** sin un tipo de excepción proporciona una manera limpia de capturar todas las excepciones, como se describió con anterioridad. Sin embargo, la tarea de capturar excepciones en clases derivadas es muy importante en otros contextos, en especial cuando creas excepciones propias.

Si quieras capturar excepciones tanto del tipo clase base como de sus derivadas, coloca primero la clase derivada en la secuencia **catch**. Esto es necesario porque el **catch** de la clase base también atrapará todas las clases derivadas. Por fortuna, esta regla es autoaplicable porque al colocar primero la clase base provoca un error en tiempo de compilación.

El siguiente programa crea dos clases excepción llamadas **ExceptA** y **ExceptB**. **ExceptA** se deriva de **Exception**. **ExceptB** se deriva de **ExceptA**. Luego, el programa lanza una excepción de cada tipo. Por cuestión de brevedad, proporciona sólo un constructor (que toma una cadena de caracteres que describe la excepción). Pero recuerda, en código comercial, tus clases de excepción personalizadas por lo regular proporcionarán los cuatro constructores definidos por **Exception**.

```
// Las excepciones derivadas deben aparecer antes que las excepciones
// de la clase base.
using System;

// Crea una excepción.
class ExceptA : Exception {
    public ExceptA(string str) : base(str) { }

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

// Crea una excepción derivada de ExceptA
class ExceptB : ExceptA { ← Observa que ExceptB deriva de ExceptA.
    public ExceptB(string str) : base(str) { }

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

class OrdenImporta {
    static void Main() {
        for(int x = 0; x < 3; x++) {
            try {
                if (x==0) throw new ExceptA("Captura una excepción ExceptA");
                else if (x==1) throw new ExceptB("Captura una excepción
                    ExceptB");
                else throw new Exception();
            }
            catch (ExceptB exc) { ←
                Console.WriteLine(exc);
            }
            catch (ExceptA exc) { ←
                Console.WriteLine(exc);
            }
            catch (Exception exc) { ←
                Console.WriteLine(exc);
            }
        }
    }
}
```

Los datos generados por el programa son:

```
Captura una excepción ExceptA  
Captura una excepción ExceptB  
System.Exception: Exception of type 'System.Exception' was thrown.  
    at OrdenImporta.Main()  
(System.Exception: Excepción de tipo 'System.Exception' fue lanzada. en OrdenImporta.Main())
```

Observa el orden de las cláusulas **catch**. Es el único orden en que pueden ocurrir. Como **ExceptB** es derivada de **ExceptA**, el **catch** para **ExceptB** debe anteceder al de **ExceptA**. De manera similar, el **catch** para **Exception** (que es la clase base para todas las excepciones) debe aparecer al final. Para comprobar por ti mismo este punto, intenta reordenar las cláusulas **catch**. Al hacerlo generarás un error en tiempo de compilación.

### Pregunta al experto

**P:** Dado que una excepción por lo regular indica un error específico, ¿por qué querría capturar una excepción de clase base?

**R:** Una cláusula **catch** que atrapa una excepción de la clase base te permite atrapar una categoría entera de excepciones, posiblemente manejarlas con un solo **catch** y evitar la duplicación de código. Por ejemplo, puedes crear un conjunto de excepciones que describan algún tipo de error de dispositivo. Si tus controladores de excepción simplemente le indican al usuario que ocurrió un error de dispositivo, entonces puedes utilizar un **catch** común para todas las excepciones de ese tipo. El controlador puede mostrar simplemente la cadena **Message**. Como el código que realiza esta tarea es el mismo para todas las excepciones, un solo **catch** puede responder a todos los errores de dispositivo.

### Prueba esto

### Añadir excepciones a la clase orden en cola

En este proyecto crearás dos clases excepción que pueden servir para las clases de orden en cola desarrolladas en la sección *Prueba esto* del capítulo 9. Indicarán las condiciones de error para una cola llena y una vacía. Tales excepciones serán lanzadas por los métodos **Put()** y **Get()**, respectivamente, cuando una de ellas ocurra. Por razones de sencillez, este ejemplo añadirá estas excepciones sólo a la clase **ColaSimple**, pero puedes incorporarlas con facilidad a las demás clases.

Para ser breves, las clases de excepción implementan sólo el constructor que actualmente se está utilizando en el programa (es el mismo que toma el argumento en forma de cadena de caracteres que describe la excepción.) Puedes intentar añadir los otros por cuenta propia a manera de ejercicio.

## Paso a paso

**1.** Crea un archivo llamado **QExcDemo.cs**.

**2.** En **QExcDemo.cs**, define las siguientes excepciones:

```
// Añade control de excepciones a las clases de orden en cola.

using System;

// Una excepción para errores de cola llena.
class ColaLlenaExcep : Exception {
    public ColaLlenaExcep(string str) : base(str) { }
    // Añade otros constructores ColaLlenaExcep aquí, si lo deseas.

    public override string ToString() {
        return "\n" + Message;
    }
}

// Una excepción para errores de cola vacía.
class ColaVacExcep : Exception {
    public ColaVacExcep(string str) : base(str) { }
    // Añade otros constructores ColaVacExcep aquí, si lo deseas.

    public override string ToString() {
        return "\n" + Message;
    }
}
```

Una excepción **ColaLlenaExcep** se genera cuando se intenta almacenar un elemento en una cola llena. Una excepción **ColaVacExcep** se genera cuando se intenta recuperar un elemento de una cola vacía.

**3.** Modifica la clase **ColaSimple** de manera que lance una excepción cuando ocurra un error, como se muestra a continuación. Añade esto a **QExcDemo.cs**.

```
// Una clase de cola simple y tamaño fijo para caracteres que utiliza
// excepciones.
class ColaSimple : ICharQ {
    char[] q; // este arreglo contiene la cola
    int putloc, getloc; // los índices put y get

    // Construye una cola vacía dado su tamaño.
    public ColaSimple(int tamaño) {
        q = new char[tamaño+1]; // reserva memoria para la cola
        putloc = getloc = 0;
    }

    // Coloca un carácter en la cola.
```

(continúa)

```
public void Put(char ch) {
    if(putloc==q.Length-1)
        throw new ColaLlenaExcep("¡Cola llena! Longitud máxima es " +
                                  (q.Length-1) + ".");
    putloc++;
    q[putloc] = ch;
}

// Recupera un carácter de la cola.
public char Get() {
    if(getloc == putloc)
        throw new ColaVacExcep("La cola está vacía.");
    getloc++;
    return q[getloc];
}
```

La adición de excepciones a **ColaSimple** permite que un error en la cola sea controlado de manera comprensible. Tal vez recuerdes que en la versión anterior de **ColaSimple** simplemente se reportaba el error. Lanzar una excepción es un medio mucho más efectivo porque permite que el código que utiliza **ColaSimple** maneje el error de manera apropiada.

4. Para probar la actualización de la clase **ColaSimple**, añade a **QExcDemo.cs** la clase **QExcDemo** que se muestra a continuación.

```
// Muestra las excepciones de cola.

class QExcDemo {
    static void Main() {
        ColaSimple q = new ColaSimple(10);
        char ch;
        int i;

        try {
            // Desborda la cola.
            for(i=0; i < 11; i++) {
                Console.WriteLine("Intenta almacenar : " +
                                  (char)('A' + i));
                q.Put((char)('A' + i));
                Console.WriteLine(" -- OK");
            }
            Console.WriteLine();
        }
        catch (ColaLlenaExcep exc) {
            Console.WriteLine(exc);
        }
        Console.WriteLine();
```

```
try {
    // Vacía la cola.
    for(i=0; i < 11; i++) {
        Console.WriteLine("Obtiene el siguiente carácter: ");
        ch = q.Get();
        Console.WriteLine(ch);
    }
}
catch (ColaVacExcep exc) {
    Console.WriteLine(exc);
}
}
```

5. Para crear el programa, debes compilar **QExcDemo.cs** con el archivo **IQChar.cs**. Recuerda que este último contiene la interfaz de cola.<sup>1</sup> Cuando ejecutas **QExcDemo** verás los siguientes datos:

```
Intenta almacenar : A -- OK
Intenta almacenar : B -- OK
Intenta almacenar : C -- OK
Intenta almacenar : D -- OK
Intenta almacenar : E -- OK
Intenta almacenar : F -- OK
Intenta almacenar : G -- OK
Intenta almacenar : H -- OK
Intenta almacenar : I -- OK
Intenta almacenar : J -- OK
Intenta almacenar : K
¡Cola llena! Longitud máxima es 10.
```

```
Obtiene el siguiente carácter: A  
Obtiene el siguiente carácter: B  
Obtiene el siguiente carácter: C  
Obtiene el siguiente carácter: D  
Obtiene el siguiente carácter: E  
Obtiene el siguiente carácter: F  
Obtiene el siguiente carácter: G  
Obtiene el siguiente carácter: H  
Obtiene el siguiente carácter: I  
Obtiene el siguiente carácter: J  
Obtiene el siguiente carácter:  
La cola está vacía.
```

---

<sup>1</sup> No olvides dejar fuera el archivo IQDemo.cs, porque contiene un método Main( ), que chocará con el declarado en QExeDemo.cs y no te permitirá compilar el programa.

## Utilizar checked y unchecked

Un cálculo aritmético puede provocar un desbordamiento. Por ejemplo, observa la siguiente secuencia:

```
byte a, b, resultado;  
a = 127;  
b = 127;  
  
resultado = (byte) (a * b);
```

Aquí, el resultado de `a` por `b` excede el rango de un valor `byte`. Así, el resultado desborda la capacidad de su tipo.

C# te permite especificar si tu código generará una excepción cuando ocurre un desbordamiento utilizando las palabras clave **checked** y **unchecked**. Para especificar si una expresión se verifica por si ocurre un desbordamiento, utiliza **checked**. Para indicar que una expresión se ignora, utiliza **unchecked**. En este caso, el resultado será truncado para acoplarlo a los límites del tipo que se está utilizando en la expresión.

La palabra clave **checked** tiene dos formatos generales. Uno valida una expresión específica y es llamado *formato de operador checked*. El otro valida un bloque de declaraciones y es llamado *formato de declaración*.

`checked (expresión)`

```
checked {  
    // declaraciones a validar  
}
```

Aquí, *expresión* es la expresión que se validará. Si la expresión validada se desborda, entonces se lanza una excepción **OverflowException**.

La palabra clave **unchecked** también tiene dos formatos generales. El primero es el formato de operador, que ignora el desbordamiento de una expresión específica. El otro ignora el desbordamiento de un bloque de declaraciones. Ambos se muestran a continuación:

`unchecked (expresión)`

```
unchecked {  
    // declaraciones donde el desbordamiento será ignorado  
}
```

Aquí, *expresión* es la expresión que será ignorada en un desbordamiento. Si una expresión ignorada se desborda, entonces se truncará.

A continuación presentamos un programa que muestra el funcionamiento de **checked** y **unchecked**.

```
// Uso de checked y unchecked.  
using System;
```

```

class CheckedDemo {
    static void Main() {
        byte a, b;
        byte result;

        a = 127;
        b = 127;           El desbordamiento en esta
                           expresión es truncado.

        try {
            result = unchecked((byte)(a * b));
            Console.WriteLine("Unchecked result: " + result);
        }
        result = checked((byte)(a * b)); // esto provoca una excepción
        Console.WriteLine("Checked result: " + result); // no se ejecutará
    }
    catch (OverflowException exc) {
        Console.WriteLine(exc);
    }
}
}

```

↓

El desbordamiento aquí genera una excepción.

Los datos generados por el programa son:

```

Unchecked result: 1
System.OverflowException: Arithmetic operation resulted in an overflow.
  at CheckedDemo.Main()
(System.OverflowException: Operación aritmética resultó en un desbordamiento en CheckedDe-
mo.Main()).

```

Como es evidente, la expresión sin validación da un resultado trunco. La expresión validada provoca una excepción.

El programa anterior mostró el uso de **checked** y **unchecked** para una sola expresión. El siguiente muestra su uso en un bloque de declaraciones.

```

// Uso de checked y unchecked con bloques de declaraciones.
using System;

class CheckedBlocks {
    static void Main() {
        byte a, b;
        byte resultado;

        a = 127;
        b = 127;

        try {
            unchecked { ← Un bloque sin validar (unchecked).
                a = 127;
                b = 127;
            }
        }
    }
}

```

```
resultado = (byte)(a * b);
Console.WriteLine("Resultado Unchecked: " + resultado);

a = 125;
b = 5;
resultado = (byte)(a * b);
Console.WriteLine("Resultado Unchecked: " + resultado);
}

checked { ← Un bloque validado (checked).
a = 2;
b = 7;
resultado = (byte)(a * b); // esto es correcto
Console.WriteLine("Resultado Checked: " + resultado);

a = 127;
b = 127;
resultado = (byte)(a * b); // esto provoca una excepción
Console.WriteLine("Resultado Checked: " + resultado); // no se
ejecutará
}
}
catch (OverflowException exc) {
    Console.WriteLine(exc);
}
}
}
```

Los datos generados por el programa son:

```
Resultado Unchecked: 1
Resultado Unchecked: 113
Resultado Checked: 14
System.OverflowException: Arithmetic operation resulted in an overflow.
at CheckedBlocks.Main()
```

Como puedes ver, el bloque sin validar da como resultado que el desbordamiento se trunque. Cuando ocurre el desbordamiento en el bloque validado, se genera una excepción.

Una razón por la que puedes utilizar **checked** y **unchecked** es que el estatus de validación o no validación es determinado por la configuración de una opción de compilación y por el mismo ambiente de ejecución. De esta manera, para algunos tipos de programa es mejor especificar explícitamente el estatus de la validación del desbordamiento.

## Pregunta al experto

**P:** ¿Cuándo debo utilizar el control de excepciones en un programa? ¿Cuándo debo crear mis propias clases de excepción personalizadas?

**R:** Como C# utiliza extensivamente excepciones para reportar errores, casi todos los programas en el mundo real harán uso del control de excepciones. Ésta es la parte del control de excepciones que resulta sencilla para la mayoría de los nuevos programadores en C#. Más difícil es decidir cuándo y cómo utilizar tus propias excepciones personalizadas. En general, hay dos maneras en que pueden ser reportados los errores: regresando valores y excepciones. ¿Cuándo es mejor utilizar una u otra? La norma debería ser poner el control de las excepciones en C#. Es verdad que regresar códigos de error es una alternativa válida en algunos casos, pero las excepciones proporcionan un medio más poderoso y estructurado para el manejo de errores. Ése es el modo en que los programadores profesionales de C# manejan los errores en su código.

## ✓ Autoexamen Capítulo 10

1. ¿Cuál es la clase superior en la jerarquía de las excepciones?
2. Explica brevemente cómo utilizar **try** y **catch**.
3. ¿Qué errores tiene este fragmento?

```
// ...
vals[18] = 10;
catch (IndexOutOfRangeException exc) {
    // maneja el error
}
```

4. ¿Qué sucede si una excepción no es capturada?
5. ¿Qué errores tiene este fragmento?

```
class A : Exception { ...
class B : A { ...
// ...
try {
    // ...
}
catch (A exc){ ... }
catch (B exc){ ... }
```

6. ¿Una excepción capturada por el **catch** interno puede regresar a un **catch** externo?
7. El bloque **finally** es el último trozo de código ejecutado antes de finalizar el programa. ¿Cierto o falso? Explica la respuesta.
8. En el ejercicio 3 de autoexamen del capítulo 6 creaste una clase **Pila**. Añade excepciones personalizadas a tu clase para reportar las condiciones de pila llena y pila vacía. Incluye los cuatro constructores de clase de excepción estándar.
9. Explica el propósito de **checked** y **unchecked**.
10. ¿Cómo se pueden capturar todas las excepciones?

# Capítulo 11

## Utilizar E/S

## Habilidades y conceptos clave

- El flujo de datos
  - Clases de flujo de datos
  - Consola E/S
  - Archivo E/S
  - Leer y escribir datos binarios
  - Archivos de acceso aleatorio
  - Convertir cadenas numéricas
- 

Desde el principio de este libro, has venido utilizando partes del sistema E/S (Entrada/Salida) de C#, como `Console.WriteLine()`, pero lo has hecho sin una explicación formal. Como el sistema E/S de C# está construido sobre una jerarquía de clases, no era posible presentar su teoría y detalles sin haber abordado primero las clases, la herencia y las excepciones. Ahora es momento de examinar con detalle la manera en que C# aborda E/S. Como se explicó en el capítulo 1, C# utiliza el sistema E/S y las clases definidas por.NET Framework. Por tanto, abordar E/S para C# es también abordarlo para el sistema E/S .NET en general.

Este capítulo examina el enfoque de C# tanto para el sistema E/S de consola como para el de archivos. Te anticipo que este sistema es muy largo. Este capítulo presenta las características más importantes y las de mayor uso, pero habrá muchos aspectos de E/S que querrás estudiar por tu cuenta. Por fortuna, el sistema E/S de C# es cohesivo y consistente; una vez que comprendas sus fundamentos, el resto es fácil de dominar.

## E/S de C# está construido sobre flujo de datos

Los programas de C# realizan tareas E/S a través del flujo o transmisión de datos. El *flujo* es un concepto abstracto que se refiere a la producción y consumo de información. El flujo está ligado a un dispositivo físico por el sistema E/S. Todo flujo se comporta de la misma manera, incluso si los dispositivos físicos a los que está vinculado son diferentes. De esta manera, las clases y los métodos E/S se pueden aplicar a una gran cantidad de dispositivos. Por ejemplo, los mismos métodos que utilizas para escribir en la consola pueden utilizarse para escribir un archivo de disco.

### Flujo de bytes y flujo de caracteres

En el nivel más bajo, todas las tareas E/S de C# operan sobre bytes. Esto tiene sentido porque muchos dispositivos están orientados a bytes cuando se trata de operaciones E/S. Pero con frecuencia, los humanos preferimos comunicarnos utilizando caracteres. Recuerda que en C#, `char` es un tipo de 16 bits y `byte` es un tipo de 8 bits. Si utilizas el juego de caracteres ASCII, entonces es fácil

convertir de **char** a **byte**; sólo ignora el bit de orden alto del valor **char**. Pero esto no funcionará con el resto de los caracteres Unicode, que necesitan ambos bytes. Por tales razones, los flujos de bytes no son perfectamente adecuados para manejar tareas E/S basadas en caracteres. Para resolver este problema, C# define varias clases para convertir un flujo de bytes en un flujo de caracteres, controlando la traducción de **byte-a-char** y **char-a-byte** automáticamente para ti.

## Flujos predefinidos

Tres flujos predefinidos, que son expuestos por las propiedades **Console.In**, **Console.Out** y **Console.Error**, están disponibles para todos los programas que utilizan la nomenclatura **System**.

**Console.Out** se refiere al flujo de salida estándar. De forma predeterminada, se utiliza la consola. Cuando invocas **Console.WriteLine()**, por ejemplo, en automático envía información a **Console.Out**. **Console.In** hace referencia al flujo de entrada estándar, que es el teclado. **Console.Error** se refiere al flujo de errores estándar, el cual también es la consola. No obstante, estos tres flujos pueden ser redireccionados a cualquier dispositivo E/S compatible. Los flujos estándar son flujos de caracteres. Así, estos flujos leen y escriben caracteres.

## Las clases de flujo

El sistema de E/S define clases tanto de flujo de bytes como de flujo de caracteres. Sin embargo, las clases de flujo de caracteres son realmente cubiertas que transforman el flujo subyacente de bytes en flujo de caracteres, manejando toda conversión de manera automática. De esta manera, los flujos de caracteres, aunque estén separados lógicamente, se construyen sobre flujos de bytes.

Todas las clases flujo están definidas dentro de la nomenclatura **System.IO**.<sup>1</sup> Para utilizar estas clases, por lo regular incluirás la siguiente declaración cerca del inicio del programa:

```
using System.IO;
```

La razón por la que no tienes que especificar **System.IO** para las entradas y salidas de la consola es porque la clase **Console** está definida en la nomenclatura **System**.

## La clase Stream

El corazón de la clase flujo (*stream*) es **System.IO.Stream**, representa un flujo de bytes y es la clase base para todas las demás clases flujo. También es abstracta, lo cual significa que no puedes inicializar un objeto **Stream** directamente. **Stream** define un conjunto de operadores estándar de flujo. La tabla 11-1 muestra algunos de los métodos más comunes definidos por **Stream**.

Varios de los métodos mostrados en la tabla 11-1 lanzarán una excepción **IOException** si ocurre un error E/S. Si se intenta realizar una operación inválida, como escribir hacia un flujo que es sólo-lectura, se lanza una excepción **NotSupportedException**. También son posibles otras excepciones, dependiendo del método específico.

Observa que **Stream** define métodos que leen y escriben datos. Sin embargo, no todos los flujos soportarán ambas operaciones porque es posible abrir flujos de sólo-lectura y sólo-escritura. De la misma manera, no todos los flujos soportarán posiciones requeridas a través de **Seek()**. Para determinar las capacidades de un flujo, utilizarás una o más propiedades de la clase **Stream**. Tales

---

<sup>1</sup> IO son las siglas de Input/Output, traducido como Entrada/Salida (E/S).

propiedades se muestran en la tabla 11-2. En esa misma tabla aparecen las propiedades **Length** y **Position**, que contienen la longitud de un flujo y su posición actual.

| Método                                          | Descripción                                                                                                                                                                    |
|-------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void Close( )                                   | Cierra un flujo.                                                                                                                                                               |
| void Flush( )                                   | Escribe el contenido de un flujo en un dispositivo físico.                                                                                                                     |
| int ReadByte( )                                 | Regresa la representación en números enteros del siguiente byte disponible de entrada. Regresa -1 cuando encuentra el final del archivo.                                       |
| int Read(byte[ ]buf, int offset, int numBytes)  | Intenta leer hasta el byte determinado por <i>numBytes</i> en <i>buf</i> comenzando en <i>buf[offset]</i> ; regresa la cantidad de datos leídos con éxito.                     |
| long Seek(long offset, SeekOrigin origin)       | Establece la posición actual en el flujo para el <i>offset</i> especificado a partir del <i>origin</i> especificado.                                                           |
| void WriteByte(byte b)                          | Escribe un solo byte hacia un flujo de salida.                                                                                                                                 |
| int Write(byte[ ]buf, int offset, int numBytes) | Escribe un subrango de bytes especificado por <i>numBytes</i> desde el arreglo <i>buf</i> , comenzando a partir de <i>buf[offset]</i> . Regresa la cantidad de bytes escritos. |

**Tabla 11-1** Una muestra de métodos definidos por **Stream**

## Las clases de flujo de bytes

Varios flujos de byte concretos son derivados de **Stream**. Aquellos que son definidos en la nomenclatura **System.IO** se muestran a continuación.

| Clase Stream          | Descripción                                                                                                        |
|-----------------------|--------------------------------------------------------------------------------------------------------------------|
| BufferedStream        | Envuelve un flujo de bytes y añade memoria. La memoria reservada proporciona en muchos casos un mejor rendimiento. |
| FileStream            | Un flujo de bytes diseñado para archivo E/S.                                                                       |
| MemoryStream          | Un flujo de bytes que utiliza memoria para almacenamiento.                                                         |
| UnmanagedMemoryStream | Un flujo de bytes que usa memoria para almacenamiento, pero no es apta para programación de lenguaje cruzado.      |

Muchas otras clases concretas de flujo que proporcionan soporte para archivos comprimidos, sockets y pipes, entre otros, son también soportadas por .NET Framework. También puedes derivar tus propias clases de flujo, pero para la gran mayoría de aplicaciones, los flujos integrados serán suficientes.

## Las clases envueltas de flujo de caracteres

Para crear un flujo de caracteres, envolverás un flujo de bytes dentro de una de las envolturas de flujo de caracteres. En la parte más alta de la jerarquía de flujo de caracteres se encuentran las clases abstractas **TextReader** y **TextWriter**. Los métodos definidos por estas dos clases abstractas están disponibles para todas sus subclases. De esta manera, forman un conjunto mínimo de funciones E/S que tendrán todos los flujos de caracteres.

| Propiedad        | Descripción                                                                                                                                                |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| bool CanRead     | Esta propiedad es verdadera si el flujo puede ser leído. Esta propiedad es sólo-lectura.                                                                   |
| bool CanSeek     | Esta propiedad es verdadera si el flujo soporta requerimientos de posición. Esta propiedad es sólo-lectura.                                                |
| bool CanTimeout  | Esta propiedad es verdadera si el flujo tiene un tiempo de expiración. Esta propiedad es sólo-lectura.                                                     |
| bool CanWrite    | Esta propiedad es verdadera si el flujo puede recibir escritura. Esta propiedad es sólo-lectura.                                                           |
| long Length      | Esta propiedad contiene la longitud del flujo. Esta propiedad es sólo-lectura.                                                                             |
| long Position    | Esta propiedad representa la posición actual del flujo. Esta propiedad es lectura/escripción.                                                              |
| int ReadTimeout  | Esta propiedad representa la longitud del tiempo antes de que ocurra una interrupción para operaciones de lectura. Esta propiedad es lectura/escripción.   |
| int WriteTimeout | Esta propiedad representa la longitud del tiempo antes de que ocurra una interrupción para operaciones de escritura. Esta propiedad es lectura/escripción. |

**Tabla 11-2** Las propiedades definidas por **Stream**

La tabla 11-3 muestra los métodos de entrada en **TextReader**. En general, estos métodos pueden lanzar una excepción **IOException** en un error. (Algunos también pueden lanzar otro tipo de excepciones.) De particular interés resulta el método **ReadLine()**, que lee una línea completa de texto, regresándola como **string**. Este método resulta de utilidad cuando se leen datos de entrada que contienen espacios incrustados.

| Método                                              | Descripción                                                                                                                                                             |
|-----------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int Peek( )                                         | Obtiene el siguiente carácter del flujo de entrada, pero no elimina ese carácter. Regresa -1 si no hay caracteres disponibles.                                          |
| int Read( )                                         | Regresa una representación en números enteros del siguiente carácter disponible en el flujo de caracteres. Regresa -1 cuando se alcanza el final del flujo.             |
| int Read(char[ ]buf, int offset, int numChars)      | Intenta leer hasta los caracteres determinados por <i>numChars</i> en <i>buf</i> comenzando en <i>buf[offset]</i> ; regresa la cantidad de caracteres leídos con éxito. |
| int ReadBlock(char[ ]buf, int offset, int numChars) | Intenta leer hasta los caracteres determinados por <i>numChars</i> en <i>buf</i> comenzando en <i>buf[offset]</i> ; regresa la cantidad de caracteres leídos con éxito. |
| string ReadLine( )                                  | Lee la siguiente línea de texto y la regresa como cadena de caracteres. Regresa null si se intenta leer un fin-de-archivo.                                              |
| string ReadToEnd                                    | Regresa todos los caracteres restantes en un flujo como cadena de caracteres.                                                                                           |

**Tabla 11-3** Los métodos de entrada definidos por **TextReader**

**TextWriter** define versiones de **Write( )** y **WriteLine( )** que muestran en pantalla todos los tipos integrados. Por ejemplo, a continuación se presentan unas cuantas versiones sobrecargadas:

| Método                     | Descripción                                                   |
|----------------------------|---------------------------------------------------------------|
| void Write(int val)        | Escribe un <b>int</b> .                                       |
| void Write(double val)     | Escribe un <b>double</b> .                                    |
| void Write(bool val)       | Escribe un <b>bool</b> .                                      |
| void WriteLine(string val) | Escribe una cadena de caracteres seguida por una nueva línea. |
| void WriteLine(uint val)   | Escribe un <b>uint</b> seguido por una nueva línea.           |
| void WriteLine(char val)   | Escribe un carácter seguido por una nueva línea.              |

Todos lanzan una excepción **IOException** si ocurre un error mientras se escribe.

Además de **Write( )** y **WriteLine( )**, **TextWriter** define los métodos **Close( )** y **Flush( )** mostrados a continuación:

```
virtual void Close()
```

```
virtual void Flush()
```

**Flush( )** provoca que sean escritos en el medio físico cualesquiera de los datos remanentes en la memoria reservada de salida. **Close( )** cierra el flujo.

Las clases **TextReader** y **TextWriter** son implementadas por varias clases de flujo basadas en caracteres, incluyendo las que se muestran a continuación. De manera que estos flujos proporcionan los métodos y las propiedades especificadas por **TextReader** y **TextWriter**.

| Clase Stream | Descripción                                                                                  |
|--------------|----------------------------------------------------------------------------------------------|
| StreamReader | Lee caracteres de un flujo de bytes. Esta clase envuelve un flujo de bytes de entrada.       |
| StreamWriter | Escribe caracteres hacia un flujo de bytes. Esta clase envuelve un flujo de bytes de salida. |
| StringReader | Lee caracteres de una cadena.                                                                |
| StringWriter | Escribe caracteres hacia una cadena.                                                         |

## Flujos binarios

Además de los flujos de bytes y caracteres, existen otras dos clases de flujo binario, que pueden ser utilizados para leer y escribir datos binarios directamente. Tales flujos son llamados **BinaryReader** y **BinaryWriter**. Analizaremos con detalle ambos más tarde en este mismo capítulo cuando abordemos el tema de archivos binarios E/S.

Ahora que comprendes el diseño general del sistema E/S, el resto del capítulo examinará las diferentes piezas con detalle, comenzando con la E/S de la consola.

## E/S de la consola

La E/S de la consola se realiza a través de los flujos estándar **Console.In**, **Console.Out** y **Console.Error**. Has venido utilizando la E/S de la consola desde el capítulo 1, por lo que ya estás familiarizado con ella. Como verás, tiene algunas capacidades adicionales.

Sin embargo, antes de comenzar es importante hacer énfasis en un punto mencionado con anterioridad en este libro: la mayoría de las aplicaciones C# del mundo real no serán programas de consola basados en texto. Lejos de eso, serán programas orientados al ambiente gráfico o componentes que dependen de una interfaz gráfica para interactuar con el usuario, o bien serán aplicaciones que se ejecutan del lado del servidor. Por lo mismo, la porción de sistemas de E/S que se relacionan con la consola no se utiliza mucho. Aunque los programas basados en texto son excelentes como ejemplos para la enseñanza y para programar utilidades breves, no se acoplan a la mayoría de las aplicaciones del mundo real.

### Ler datos de entrada de la consola

**Console.In** es una instancia de **TextReader**, y puedes utilizar los métodos y propiedades definidos por **TextReader** para accesarla. Sin embargo, por lo regular utilizarás los métodos proporcionados por **Console**, que leen automáticamente de **Console.In**. **Console** define dos métodos de entrada: **Read()** y **ReadLine()**.

Para leer un solo carácter, utiliza el método **Read()**, como se muestra a continuación:

```
static int Read()
```

Este método fue presentado en el capítulo 3. Regresa el siguiente carácter leído de la consola. El carácter es regresado como un **int**, que debe ser transformado a **char**. Regresa -1 en caso de error. Este método lanzará una excepción **IOException** en caso de falla. Recuerda que **Read()** se almacena en la memoria, por lo que debes oprimir la tecla ENTER antes de que cualquier carácter que hayas escrito sea enviado a tu programa.

Para leer una cadena de caracteres, utiliza el método **ReadLine()**, como se muestra aquí:

```
static string ReadLine()
```

**ReadLine()** lee caracteres hasta que oprimes la tecla ENTER y los regresa como un objeto **string**. Este método también lanzará una excepción **IOException** en caso de falla.

He aquí un programa que muestra la lectura de una línea de caracteres de **Console.In**:

```
// Datos provenientes de la consola utilizando ReadLine().
using System;

class LeeChars {
    static void Main() {
        string str;

        Console.WriteLine("Escribe algunos caracteres.");
        str = Console.ReadLine(); ← Lee una cadena proveniente del teclado.
        Console.WriteLine("Escribiste: " + str);
    }
}
```

Un ejemplo de los datos generados por el programa:

```
Escribe algunos caracteres.  
Ésta es una prueba.  
Escribiste: Ésta es una prueba.
```

Aunque los métodos **Console** representan la manera más fácil de leer datos desde la consola, puedes invocar métodos del **TextReader** subyacente, que está disponible a través de **Console.In**. Por ejemplo, a continuación presentamos una versión modificada del programa anterior que utiliza métodos definidos por **TextReader**:

```
/* Lee un arreglo de bytes desde el teclado, utilizando  
   Console.In directamente. */  
using System;  
  
class LeeChars2 {  
    static void Main() {  
        string str;  
  
        Console.WriteLine("Escribe algunos caracteres.");  
  
        str = Console.In.ReadLine(); ←—— Lee explícitamente de Console.In.  
        Console.WriteLine("Escribiste: " + str);  
    }  
}
```

Observa que ahora **ReadLine()** invoca directamente a **Console.In**. El punto clave aquí es que si necesitas acceso a los métodos definidos por **TextReader** que subyace en **Console.In**, invocarás esos métodos como se muestra en el ejemplo.

## Escribir datos de salida de la consola

**Console.Out** y **Console.Error** son objetos del tipo **TextWriter**. Enviar datos de salida desde la consola se realiza con mayor facilidad con los métodos **Write()** y **WriteLine()**, con los cuales ya estás familiarizado. Existen versiones de estos métodos que manejan datos de salida para cada tipo integrado. **Console** define sus propias versiones para **Write()** y **WriteLine()** de manera que puedan ser invocadas directamente en **Console**, como lo has estado haciendo a lo largo de este libro. Sin embargo, puedes invocar estos métodos (y otros) desde **TextWriter** que subyace en **Console**.

**Out** y **Console.Error**, si así lo decides.

He aquí un programa que muestra el proceso de escritura hacia **Console.Out** y **Console.Error**. De forma predeterminada, ambos envían los datos de salida a la consola.

```
// Escribir hacia Console.Out y Console.Error.  
using System;  
  
class ErrOut {  
    static void Main() {  
        int a=10, b=0;  
        int result;
```

```

Console.Out.WriteLine("Esto generará una excepción.");
try {
    result = a / b; // genera una excepción
} catch (DivideByZeroException exc) {
    Console.Error.WriteLine(exc.Message);
}
}

```

↑  
Escribe a **Console.Out**.  
↑  
Escribe a **Console.Error**.

Los datos generados por el programa son:

Esto generará una excepción.  
Attempted to divide by zero.  
(Intento de dividir entre cero)

En ocasiones, los programadores novatos se confunden sobre cuándo utilizar **Console.Error**. Como **Console.Out** y **Console.Error** escriben sus datos de salida en la consola, ¿por qué se consideran flujos diferentes? La respuesta se encuentra en el hecho de que los flujos estándar pueden ser redireccionados hacia otros dispositivos. Por ejemplo, **Console.Error** puede redireccionarse para que escriba sobre un archivo de disco, en lugar de hacerlo en la pantalla. De esta manera, es posible dirigir los datos de salida del error a una bitácora, por ejemplo, sin afectar la salida de la consola. A la inversa, si los datos de salida de la consola son redireccionados y no sucede lo mismo con los datos de salida del error, aparecerán mensajes de error en la consola, donde puedan verse. Examinaremos la redirección más adelante, después de describir la E/S de archivos.

## Pregunta al experto

**P:** Como ya explicó, **Read()** es de memoria reservada. Esto significa que almacena en memoria las teclas que se oprimen en el teclado y no las envía hasta que se oprime la tecla **ENTER**. ¿C# soporta un método interactivo de entrada que haga el regreso en cuanto se oprime la tecla?

**R:** ¡Sí! Desde la versión 2.0, .NET Framework ha incluido un método en **Console** que te permite leer golpes de teclas individuales directamente desde el teclado sin tener que reservar memoria. Este método se llama **.ReadKey()**. Cuando se invoca, espera hasta que se oprime una tecla. Cuando esto ocurre, **.ReadKey()** regresa la tecla oprimida inmediatamente. No es necesario que oprimas **ENTER**. De esta manera, **.ReadKey()** permite que el tecleo sea leído y procesado en tiempo real. **.ReadKey()** tiene estos dos formatos:

```

static ConsoleKeyInfo ReadKey()
static ConsoleKeyInfo ReadKey(bool noMostrar)

```

El primero espera a que se oprima la tecla. Cuando esto ocurre, regresa el símbolo de la tecla oprimida y también despliega su valor en pantalla. El segundo también espera a que se oprima una tecla. Sin embargo, si *noMostrar* es verdadero, entonces el valor de la tecla no se muestra. Si *noMostrar* es falso, el valor de la tecla aparece en el monitor.

(continúa)

**.ReadKey( )** regresa información sobre las teclas oprimidas en un objeto de tipo **ConsoleKeyInfo**, que es una estructura. Contiene las siguientes propiedades sólo-lectura:

char KeyChar

ConsoleKey Key

ConsoleModifiers Modifiers

**KeyChar** contiene el equivalente **char** del carácter que se oprimió. **Key** contiene un valor de la enumeración **ConsoleKey**, que es una enumeración de todas las teclas del teclado. **Modifiers** describe cuáles modificadores del teclado como ALT, CTRL o SHIFT fueron oprimidos, si se oprimió alguno, cuando se oprimieron las teclas. Estos modificadores están representados por la enumeración **ConsoleModifiers**, que tiene los valores **Control**, **Shift** y **Alt**. Más de un valor puede estar presente en **Modifiers**.

## FileStream y E/S de archivos orientados a byte

El sistema E/S proporciona clases que te permiten leer y escribir archivos. Por supuesto, el tipo más común de archivo es el de disco. A nivel de sistema operativo, todos los archivos son orientados a bytes. Como era de esperarse, hay métodos que leen y escriben bytes de y hacia un archivo. Por lo mismo, leer y escribir archivos utilizando flujos de bytes es una tarea común. También puedes envolver flujos de archivo orientados a bytes dentro de objetos orientados a caracteres. Las operaciones de archivo basadas en caracteres son útiles cuando se almacena el texto. Los flujos de carácter son abordados más tarde en este mismo capítulo. En este momento estudiaremos la E/S orientada a bytes.

Para crear un flujo orientado a bytes adjunto a un archivo, utilizarás la clase **FileStream**, que se deriva de **Stream** y contiene toda su funcionalidad.

Recuerda que las clases de flujo, incluyendo **FileStream**, son definidas en **System.IO**, por lo que generalmente incluirás

```
using System.IO;
cerca del inicio de los programas que las utilicen.
```

## Abrir y cerrar un archivo

Para crear un flujo de bytes adjunto a un archivo, crea un objeto **FileStream**. Éste define varios constructores; tal vez el más utilizado es el que se muestra a continuación:

**FileStream(string nombrearchivo, FileMode modo)**

Aquí, *nombrearchivo* especifica el nombre del archivo que se va a abrir, que puede incluir una especificación completa de la ruta de acceso. El parámetro *modo* especifica la manera en que se abrirá el archivo. Debe ser uno de los valores definidos por la enumeración **FileMode**. Estos valores se muestran en la tabla 11-4. En general, este constructor abre un archivo para acceso

| Valor                 | Descripción                                                                                    |
|-----------------------|------------------------------------------------------------------------------------------------|
| FileMode.Append       | Los datos de salida son adjuntados al final del archivo.                                       |
| FileMode.Create       | Crea un nuevo archivo de salida. Todo archivo preexistente con el mismo nombre será destruido. |
| FileMode.CreateNew    | Crea un nuevo archivo de salida. El archivo no debe existir previamente.                       |
| FileMode.Open         | Abre un archivo preexistente.                                                                  |
| FileMode.OpenOrCreate | Abre un archivo si existe, o crea un archivo si todavía no existe.                             |
| FileMode.Truncate     | Abre un archivo preexistente pero reduce su longitud a cero.                                   |

**Tabla 11-4** Los valores **FileMode**

lectura/escritura. La excepción es cuando el archivo se abre utilizando **FileMode.Append**. En este caso, el archivo es sólo-lectura.

Si ocurre una falla cuando se intenta abrir un archivo, se lanzará una excepción. Si el archivo no puede abrirse porque no existe, se lanzará la excepción **FileNotFoundException**. Si el archivo no puede abrirse debido a alguna falla de E/S, se lanzará una excepción **IOException**. Otras posibles excepciones son **ArgumentNullException** (el nombre del archivo es nulo), **ArgumentException** (el nombre de archivo es inválido), **ArgumentOutOfRangeException** (el modo es inválido), **SecurityException** (el usuario no tiene permisos de acceso), **PathTooLongException** (el nombre de archivo/ruta de acceso es demasiado largo), **NotSupportedException** (el nombre de archivo especifica un dispositivo sin soporte) y **DirectoryNotFoundException** (la especificación del directorio es inválida).

Las excepciones **PathTooLongException**, **DirectoryNotFoundException** y **FileNotFoundException** son subclases de **IOException**. Por ello, es posible capturar las tres capturando **IOException**.

El siguiente ejemplo muestra una manera de abrir el archivo **test.dat** para datos de entrada:

```
FileStream fin;
try {
    fin = new FileStream("test", FileMode.Open);
}
catch (IOException exc) {
    Console.WriteLine(exc.Message);
    // Maneja el error.
}
catch(Exception exc { // capta cualquier otra excepción.
    Console.WriteLine(exc.Message);
    // Maneja el error.
}
```

Aquí, la primera cláusula **catch** maneja situaciones en las cuales el archivo no es encontrado, la ruta de acceso es demasiado larga, el directorio no existe o bien han ocurrido otros errores de E/S. El segundo **catch**, que es una cláusula “atrapa todo” para todo tipo de excepciones, maneja los demás posibles errores (posiblemente por reenviar la excepción). También debes verificar cada error

individualmente, reportando de manera más específica el problema que ocurre y tomando medidas directas para remediar ese error en particular. Con fines de sencillez, los errores que se presentan en este libro capturarán únicamente errores **IOException**, pero tu código del mundo real tal vez necesite controlar otras posibles excepciones, dependiendo de las circunstancias.

### **NOTA**

Para mantener el código sencillo, los ejemplos de este capítulo capturarán únicamente excepciones **IOException**, pero tu propio código tal vez necesite controlar otras excepciones posibles o manejar las excepciones E/S de manera individual.

Como se mencionó, a excepción de cuando se especifica  **FileMode.Append**, el constructor **FileStream** que se acaba de describir abre un archivo con acceso lectura/escritura. Si deseas restringir el acceso a sólo-lectura o sólo-escritura, utiliza el siguiente constructor en su lugar:

`FileStream(string nombrearchivo, FileMode modo, FileAccess cómo)`

Como en el caso anterior, *nombrearchivo* especifica el nombre del archivo que se abrirá y *modo* especifica la manera en que se abrirá el archivo. El valor que se transmite en *cómo* determina la manera en que el archivo puede ser accesado. Debe ser uno de los valores definidos por la enumeración **FileAccess** mostrados a continuación:

`FileAccess.Read`

`FileAccess.Write`

`FileAccess.ReadWrite`

Por ejemplo, esto abre un archivo sólo-lectura:

```
FileStream fin = new FileStream("test.dat", FileMode.Open,  
                                FileAccess.Read);
```

Cuando hayas terminado de utilizar un archivo, debes cerrarlo invocando **Close()**. Su formato general es:

`void Close()`

Cerrar un archivo libera los recursos del sistema reservados para ese archivo, permitiendo que los utilice otro archivo. Como punto de interés, **Close()** funciona invocando **Dispose()**, que de hecho es el que libera los recursos.

### **NOTA**

La declaración **using**, que se estudiará en el capítulo 15, ofrece una manera de cerrar automáticamente un archivo cuando ya no se le necesita, y este enfoque es aplicable a una gran variedad de situaciones. Sin embargo, para ilustrar con claridad los fundamentos del manejo de archivos, en este capítulo invocará de manera explícita el método **close()** en todos los casos.

## Leer bytes de un **FileStream**

**FileStream** define dos métodos que leen bytes de un archivo: **ReadByte()** y **Read()**. Para leer un solo byte de un archivo, utiliza **ReadByte()**, cuyo formato general es:

```
int ReadByte();
```

Cada vez que es invocado, lee un solo byte del archivo y lo regresa como un valor entero. Regresa -1 cuando encuentra el final del archivo. Las posibles excepciones incluyen **NotSupportedException** (el flujo no está abierto para datos de entrada) y **ObjectDisposedException** (el flujo está cerrado).

Para leer un bloque de bytes utiliza **Read()**, que tiene el siguiente formato general:

```
int Read(byte[ ] buf, int offset, int numBytes)
```

**Read()** intenta leer hasta el byte especificado por *numBytes* en *buf* comenzando en *buf[offset]*. Regresa la cantidad de bytes leídos con éxito. Se lanza una excepción **IOException** si ocurre un error de E/S. Muchos otros tipos de excepción son posibles, incluyendo **NotSupportedException**, que se lanza si el flujo no soporta lectura.

El siguiente programa utiliza **ReadByte()** para ingresar y presentar el contenido de un archivo, cuyo nombre se especifica como un argumento de comando de línea. Transforma los valores regresados por **ReadByte()** a tipo **char**, permitiendo mostrarlos como caracteres ASCII. Observa que el programa controla dos errores que pueden ocurrir cuando el programa se ejecuta por primera vez: que no se localice el archivo especificado o que el usuario olvide incluir el nombre del archivo.

```
/* Presenta un archivo.
```

Para utilizar este programa, especifica el nombre del archivo que quieras ver.

Por ejemplo, para ver un archivo llamado TEST.CS, utiliza la siguiente línea de comando.

```
ShowFile TEST.CS
*/
using System;
using System.IO;

class ShowFile {
    static void Main(string[] args) {
        int i;
        FileStream fin;

        if(args.Length != 1) {
            Console.WriteLine("Uso: ShowFile Archivo");
            return;
        }

        try {
            fin = new FileStream(args[0], FileMode.Open);
        } catch (IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }
    }
}
```

```

// Lee bytes hasta encontrar el fin del archivo, End of File (EOF).
do {
    try {
        i = fin.ReadByte(); ← Lee el archivo.
    } catch (IOException exc) {
        Console.WriteLine(exc.Message);
        break;
    }
    if(i != -1) Console.Write((char)i);
} while(i != -1); ← Cuando i es igual a -1, se ha alcanzado el fin del archivo.

fin.Close();
}
}

```

## Pregunta al experto

**P:** He notado que `ReadByte()` regresa `-1` cuando se ha alcanzado el fin del archivo, pero no tiene un valor de regreso específico para un error de archivo. ¿Por qué?

**R:** En C# los errores se representan con excepciones. Así, si `ReadByte()`, o cualquier otro método E/S, no lanza una excepción, quiere decir que se ha ejecutado sin errores. Ésta es una manera de manejar errores E/S más limpia que utilizar códigos de error.

## Escribir en un archivo

Para escribir un byte en un archivo, utiliza el método `WriteByte()`. Su formato más sencillo es:

`void WriteByte(byte val)`

Este método escribe en el archivo el byte especificado por `val`. Si el flujo subyacente no está abierto para recibir datos, se lanza una excepción `NoSupportedException`. Si el flujo está cerrado, se lanza la excepción `ObjectDisposedException`.

Puedes escribir un arreglo de bytes en un archivo invocando `Write()`, como se muestra a continuación:

`int Write(byte[ ]buf, int offset, int numBytes)`

`Write()` escribe en un archivo los bytes especificados por `numBytes` del arreglo `buf`, comenzando por `buf[offset]`. Regresa el número de bytes escritos. Si ocurre un error durante la escritura, se lanza una excepción `IOException`. Si el flujo subyacente no está abierto para recibir datos, se genera un error `NotSupportedException`. También es posible que se generen otras excepciones.

Como es posible que sepas, cuando se realiza una salida de archivo, por lo común, no son inmediatamente escritos en el dispositivo físico. En lugar de ello, los datos de salida son almacenados en memoria por el sistema operativo hasta que una cierta cantidad de datos puedan ser escritos al mismo tiempo. Esto mejora la eficiencia del sistema.

Por ejemplo, los archivos de disco son organizados en sectores, que pueden ser de 128 bytes en adelante. Por lo regular, los datos de salida son almacenados en la memoria hasta que pueda ser escrito un sector completo. Sin embargo, si quieres que los datos se escriban en el dispositivo físico, se haya completado o no el requerimiento de espacio en memoria, puedes invocar **Flush()**, como se muestra a continuación:

```
void Flush()
```

Se lanza una excepción **IOException** en caso de falla. Si el flujo estaba cerrado en el momento de la invocación, se lanza una excepción **ObjectDisposedException**.

Una vez que hayas terminado con el archivo de salida, debes recordar cerrarlo con **Close()**. Haciéndolo te aseguras de que cualquier dato de salida remanente en el buffer del disco realmente se haya escrito en el mismo. No es necesario invocar **Flush()** antes de cerrar el archivo.

El siguiente ejemplo copia un archivo. Los nombres de los archivos fuente y destino se especifican en la línea de comando.

```
/*
Copia un archivo.

Para usar este programa, especifica el nombre
del archivo fuente y el archivo destino.
Por ejemplo, para copiar un archivo llamado PRIMERO.TXT
a un archivo llamado SEGUNDO.TXT, utiliza la siguiente
línea de comando.

CopyFile PRIMERO.TXT SEGUNDO.TXT
*/
using System;
using System.IO;

class CopiaArchivo {
    static void Main(string[] args) {
        int i;
        FileStream fin;
        FileStream fout;

        if(args.Length != 2) {
            Console.WriteLine("Uso: CopiaArchivo De: Hacia:");
            return;
        }

        // Abre archivo de entrada.
        try {
            fin = new FileStream(args[0], FileMode.Open);
        } catch (IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }
```

```
// Abre archivo de salida.  
try {  
    fout = new FileStream(args[1], FileMode.Create);  
} catch (IOException exc) {  
    Console.WriteLine(exc.Message);  
    fin.Close();  
    return;  
}  
  
// Copia archivo.  
try {  
    do {  
        i = fin.ReadByte();  
        if(i != -1) fout.WriteByte((byte)i); ← Lee bytes de un archivo  
    } while (i != -1);  
} catch (IOException exc) {  
    Console.WriteLine(exc.Message);  
}  
  
fin.Close();  
fout.Close();  
}
```

## E/S de un archivo basado en caracteres

Aunque el manejo de archivos orientados a bytes es muy común, es posible utilizar flujos basados en caracteres para este propósito. La ventaja del flujo de caracteres es que operan directamente sobre caracteres Unicode. Así, si quieras almacenar texto Unicode, los flujos de caracteres son sin duda tu mejor opción. En general, para realizar operaciones de archivo basadas en caracteres, en volverás un **FileStream** dentro de un **StreamReader** o un **StreamWriter**. Estas clases convierten automáticamente un flujo de bytes en uno de caracteres y viceversa.

Recuerda, a nivel sistema operativo, un archivo consiste en un conjunto de bytes. Utilizar **StreamReader** o **StreamWriter** no modifica ese hecho.

**StreamWriter** se deriva de **TextWriter**. **StreamReader** deriva de **TextReader**. Así, **StreamWriter** y **StreamReader** tienen acceso a los métodos y propiedades definidos por sus clases base.

## Utilizar StreamWriter

Para crear un flujo de salida basado en caracteres, envuelve un objeto **Stream** (como **FileStream**) dentro de **StreamWriter**. Esta clase define varios constructores. Uno de los más populares es el que se muestra a continuación:

**StreamWriter(Stream flujo)**

Aquí, *flujo* es el nombre de un flujo abierto. Este constructor lanza una excepción **ArgumentException** si el flujo especificado no está abierto para datos de salida y una excepción **ArgumentNullException** si *flujo* es nulo. Una vez creado, un **StreamWriter** automáticamente maneja la conversión de caracteres a bytes.

A continuación presentamos una utilidad sencilla teclado-a-disco que lee líneas del texto insertado con el teclado y lo escribe en un archivo llamado **test.txt**. El texto es leído hasta que el usuario ingresa la palabra “stop” (alto). La utilidad utiliza un **FileStream** envuelto en un **StreamWriter** para sacar el archivo.

```
// Una utilidad sencilla teclado-a-disco que muestra el funcionamiento de
// StreamWriter.
using System;
using System.IO;

class TecladoADisco {
    static void Main() {
        string str;
        FileStream fout;

        try {
            fout = new FileStream("test.txt", FileMode.Create);
        }
        catch(IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }
        StreamWriter fstr_out = new StreamWriter(fout); ← Crea un
   StreamWriter.

        Console.WriteLine("Inserta texto ('stop' para concluir).");
        do {
            Console.Write(": ");
            str = Console.ReadLine();

            if (str != "stop") {
                str = str + "\r\n"; // añade nueva línea
                try {
                    fstr_out.WriteLine(str); ← Escribe líneas a un archivo.
                } catch (IOException exc) {
                    Console.WriteLine(exc.Message);
                    break;
                }
            }
        } while (str != "stop");

        fstr_out.Close();
    }
}
```

En algunos casos puedes abrir archivos directamente utilizando **StreamWriter**. Para hacerlo, utiliza uno de los siguientes constructores:

`StreamWriter(string nombrearchivo)`

`StreamWriter(string nombrearchivo, bool banderaAnexa)`

Aquí, *nombrearchivo* especifica el nombre del archivo que será abierto, que puede incluir la especificación completa de la ruta de acceso. En el segundo formato, si *banderaAnexa* es verdadera, entonces los datos de salida son anexados al final del archivo existente. De lo contrario, los datos de salida sobrescriben el archivo especificado. En ambos casos, si el archivo no existe, se crea. Además, ambos formatos lanzan una excepción **IOException** si ocurre un error de E/S. También es posible que surjan otras excepciones.

A continuación presentamos una nueva versión del programa teclado-a-disco que utiliza **StreamWriter** para abrir el archivo de salida.

```
// Abre un archivo utilizando StreamWriter.  
using System;  
using System.IO;  
  
class TecladoADisco {  
    static void Main() {  
        string str;  
        StreamWriter fstr_out;  
  
        try {  
            fstr_out = new StreamWriter("test.txt"); ← Abre un archivo utilizando  
        } catch (IOException exc) { ← StreamWriter.  
            Console.WriteLine(exc.Message);  
            return;  
        }  
  
        Console.WriteLine("Inserta texto ('stop' para concluir).");  
        do {  
            Console.Write(": ");  
            str = Console.ReadLine();  
  
            if (str != "stop") {  
                str = str + "\r\n"; // añade una nueva línea  
                try {  
                    fstr_out.Write(str);  
                } catch (IOException exc) {  
                    Console.WriteLine(exc.Message);  
                    break;  
                }  
            }  
        } while (str != "stop");  
  
        fstr_out.Close();  
    }  
}
```

## Utilizar StreamReader

Para crear un flujo de entrada basado en caracteres, envuelve un flujo de bytes dentro de un **StreamReader**. Esta clase define varios constructores. A continuación presentamos uno muy utilizado:

`StreamReader(Stream flujo)`

Aquí, *stream* es el nombre de un flujo abierto. Este constructor lanza una excepción **ArgumentNullException** si *stream* es nulo y un **ArgumentException** si *stream* no está abierto para datos de entrada. Una vez creado, un **StreamReader** automáticamente manejará la conversión de bytes a caracteres.

El siguiente programa utiliza **StreamReader** para crear una utilidad sencilla disco-a-monitor, que lee línea por línea un texto llamado **test.txt** y muestra su contenido en el monitor. Así, es un complemento de la utilidad teclado-a-disco mostrada en la sección anterior.

```
// Una utilidad sencilla disco-a-monitor que muestra el funcionamiento de
FileReader.

using System;
using System.IO;

class DiscoAMonitor {
    static void Main() {
        FileStream fin;
        string s;

        try {
            fin = new FileStream("test.txt", FileMode.Open);
        }
        catch (IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }

        StreamReader fstr_in = new StreamReader(fin);

        try {
            while ((s = fstr_in.ReadLine()) != null) { ← Lee las líneas del archivo y
                Console.WriteLine(s);
            }
        } catch(IOException exc) {
            Console.WriteLine(exc.Message);
        }

        fstr_in.Close();
    }
}
```

Observa cómo se determina el fin del archivo. Cuando la referencia regresada por **ReadLine()** es nula, se ha alcanzado el fin del archivo.

Lo mismo que en **StreamWriter**, en algunos casos puedes abrir directamente el archivo utilizando **StreamReader**. Para hacerlo, utiliza el constructor:

```
StreamReader(string nombrearchivo)
```

Aquí, *nombrearchivo* especifica el nombre del archivo que se abrirá, que puede incluir la especificación de la ruta de acceso completa. El archivo debe existir; de lo contrario, se lanzará una excepción **FileNotFoundException**. Si *nombrearchivo* es nulo, entonces se lanza una excepción **ArgumentNullException**. Si *nombrearchivo* es una cadena vacía, se lanza una excepción **ArgumentException**. También es posible que ocurran excepciones **IOException** y **DirectoryNotFoundException**.

### Pregunta al experto

**P:** He escuchado que puedo especificar una “codificación de carácter” cuando se abre un **StreamReader** o un **StreamWriter**. ¿Qué es una codificación de carácter y cuándo debo utilizarla?

**R:** **StreamReader** y **StreamWriter** convierten bytes a caracteres y viceversa basados sobre una *codificación de caracteres* que especifica cómo ocurre la transformación. De forma predeterminada, C# utiliza la codificación UTF-8, que es compatible con ASCII. Para especificar otra codificación, debes utilizar versiones sobrecargadas de los constructores **StreamReader** y **StreamWriter** que incluyan parámetros de codificación. En general, necesitarás especificar una codificación de caracteres sólo bajo circunstancias extraordinarias.

### Redireccionar los flujos estándar

Como se mencionó con anterioridad, los flujos estándar, como **Console.In**, pueden redireccionarse. Por mucho, la redirección más común es hacia un archivo. Cuando se redirige un flujo estándar, los datos de salida y de entrada son automáticamente dirigidos a un nuevo flujo, desviando los dispositivos de forma predeterminada. Al redireccionar los flujos estándar, tu programa puede leer comandos de un archivo de disco, crear bitácoras e incluso leer datos de entrada de una conexión de red.

El redireccionamiento de los flujos estándar puede realizarse de dos maneras. La primera, cuando ejecutas un programa en la línea de comandos, puedes utilizar los operadores < y > para redireccionar **Console.In** y **Console.Out**, respectivamente. Por ejemplo, dado este programa:

```
using System;

class Test {
    static void Main() {
        Console.WriteLine("Ésta es una prueba.");
    }
}
```

Ejecutar el programa de esta manera:

Test > log

provocará que la línea “Ésta es una prueba.” sea escrita a un archivo llamado **log**. Los datos de entrada pueden ser redireccionados de la misma manera. Lo que se debe recordar cuando los datos de entrada son redireccionados es asegurarse de especificar una fuente de datos de entrada que contenga el contenido suficiente para satisfacer las demandas del programa. De lo contrario, el programa quedará suspendido.

Los operadores de redirección de la línea de comandos < y > no son parte de C#, pero son proporcionados por el sistema operativo. De esta manera, si tu ambiente soporta redirección E/S (como en el caso de Windows), puedes redireccionar datos de entrada y de salida sin hacer cambios en tu programa. No obstante, existe una segunda manera de redireccionar los flujos estándar que está bajo el control de tu programa. Para hacerlo, debes utilizar los métodos **SetIn()**, **SetOut()** y **SetError()** que se muestran a continuación y que forman parte de **Console**:

```
static void SetIn(TextReader entrada)  
static void SetOut(TextWriter salida)  
static void SetError(TextWriter salida)
```

De esta manera, para redireccionar una entrada, invoca **SetIn()**, especificando el flujo deseado. Puedes utilizar cualquier flujo de entrada siempre y cuando derive de **TextReader**. Para redireccionar una salida, especifica el flujo derivado de **TextWriter**. Por ejemplo, para redireccionar datos de salida hacia un archivo, utiliza **StreamWriter**. El siguiente programa muestra un ejemplo:

```
// Redirecciona Console.Out  
using System;  
using System.IO;  
  
class Redirecciona {  
    static void Main() {  
        StreamWriter log_out;  
  
        try {  
            log_out = new StreamWriter("logfile.txt");  
        }  
        catch (IOException exc) {  
            Console.WriteLine(exc.Message);  
            return;  
        }  
  
        // Redirecciona la salida estándar hacia logfile.txt.  
        Console.SetOut(log_out); ← Redirecciona Console.Out.  
  
        try {  
            Console.WriteLine("Éste es el inicio del archivo log.");  
            for(int i=0; i<10; i++) Console.WriteLine(i);  
            Console.WriteLine("Éste es el final del archivo log.");  
        }
```

```
        } catch (IOException exc) {
            Console.WriteLine(exc.Message);
        }

        log_out.Close();
    }
}
```

Cuando ejecutas este programa, no verás ningún dato de salida en la pantalla. Sin embargo, el archivo **logfile.txt** contendrá lo siguiente:

Éste es el inicio del archivo log.

0  
1  
2  
3  
4  
5  
6  
7  
8  
9

Éste es el final del archivo log.

Tal vez quieras experimentar por tu cuenta redireccionar otros flujos integrados.

### Prueba esto

### Crear una utilidad para comparar archivos

Este ejemplo desarrolla una simple pero ventajosa utilidad para comparar archivos. Funciona abriendo los dos archivos que se van a comparar, para después leer y comparar cada conjunto de bytes correspondientes. Si se encuentra una discordancia, los archivos son distintos. Si el final de cada archivo se alcanza al mismo tiempo y no se encontraron discordancias, entonces los archivos son iguales.

## Paso a paso

1. Crea un archivo llamado **CompArchivos.cs**.
2. En **CompArchivos.cs** añade el siguiente programa:

```
/*
Compara dos archivos.

Para utilizar este programa especifica los nombres
de los archivos a comparar en la línea de comandos.

Por ejemplo:
CompArchivos PRIMERO.TXT SEGUNDO.TXT
```

```
/*
using System;
using System.IO;

class CompArchivos {
    static void Main(string[] args) {
        int i=0, j=0;
        FileStream f1;
        FileStream f2;

        if(args.Length != 2) {
            Console.WriteLine("Uso: CompArchivos Archivo1 Archivo2");
            return;
        }

        // Abre el primer archivo.
        try {
            f1 = new FileStream(args[0], FileMode.Open);
        } catch(IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }

        // Abre el segundo archivo.
        try {
            f2 = new FileStream(args[1], FileMode.Open);
        } catch(IOException exc) {
            Console.WriteLine(exc.Message);
            f1.Close();
            return;
        }

        // Compara los archivos.
        try {
            do {
                i = f1.ReadByte();
                j = f2.ReadByte();
                if(i != j) break;
            } while(i != -1 && j != -1);

            if(i != j)
                Console.WriteLine("Archivos diferentes.");
            else
                Console.WriteLine("Archivos iguales.");
        } catch(IOException exc) {
            Console.WriteLine(exc.Message);
        }
    }
}
```

```
    f1.Close();
    f2.Close();
}
```

- 3.** Para probar **CompArchivos**, primero copia **CompArchivos.cs** a un archivo llamado **temp**. Luego, ejecuta la siguiente línea de comando:

CompArchivos CompArchivos.cs temp

- 4.** El programa reportará que los archivos son iguales. Ahora, compara **CompArchivos.cs** con **CopiaArchivo.cs** (mostrado anteriormente) utilizando esta línea de comando:

CompArchivos CompArchivos.cs CopiaArchivo.cs

Estos archivos son diferentes, y **CompArchivos** reportará este hecho.

- 5.** Intenta mejorar **CompArchivos** por tu cuenta con varias opciones. Por ejemplo, añade una opción que ignore las mayúsculas. Otra idea es hacer que **CompArchivos** muestre la posición dentro del archivo donde los archivos difieren.

---

## Leer y escribir datos binarios

Hasta ahora sólo hemos leído y escrito bytes o caracteres, pero es posible (de hecho es común) leer y escribir otros tipos de datos. Por ejemplo, es posible que quieras crear un archivo que contenga tipos **int**, **double** o **short**. Para leer y escribir valores binarios de los tipos integrados en C#, debes utilizar **BinaryReader** (lector binario) y **BinaryWriter** (escritor binario). Cuando utilizas estos flujos, es importante comprender que estos datos son leídos y escritos usando su formato binario interno, no su forma de texto comprensible para los humanos.

### Escritor binario

Un **BinaryWriter** es una envoltura para un flujo de bytes que maneja la escritura de datos binarios. Su constructor más utilizado se muestra a continuación:

`BinaryWriter(Stream flujoSalida)`

Aquí, *flujoSalida* es el flujo hacia el cual se escribirán los datos. Para escribir datos de salida a un archivo, puedes utilizar el objeto creado por **FileStream** para este parámetro. Si *flujoSalida* es nulo, se lanzará una excepción **ArgumentNullException**. Si *flujoSalida* no ha sido abierto para escritura, se lanzará una excepción **ArgumentException**.

**BinaryWriter** define métodos que pueden escribir todos los tipos integrados de C#. Muchos de ellos se muestran en la tabla 11-5. **BinaryWriter** también define métodos estándar **Close()** y **Flush()** que funcionan de la misma manera como se describió con anterioridad.

| Método                   | Descripción                            |
|--------------------------|----------------------------------------|
| void Write(sbyte val)    | Escribe un byte con negativos.         |
| void Write(byte val)     | Escribe un byte sin negativos.         |
| void Write(byte[ ] buf)  | Escribe un arreglo de bytes.           |
| void Write(short val)    | Escribe un entero short.               |
| void Write(ushort val)   | Escribe un entero short sin negativos. |
| void Write(int val)      | Escribe un entero.                     |
| void Write(uint val)     | Escribe un entero sin negativos.       |
| void Write(long val)     | Escribe un entero long.                |
| void Write(ulong val)    | Escribe un entero long sin negativos.  |
| void Write(float val)    | Escribe un float.                      |
| void Write(double val)   | Escribe un double.                     |
| void Write(char val)     | Escribe un carácter.                   |
| void Write(char [ ] buf) | Escribe un arreglo de caracteres.      |
| void Write(string val)   | Escribe una línea de caracteres.       |

**Tabla 11-5** Métodos de salida de mayor uso definidos por **BinaryWriter**

## Lector binario

Un **BinaryReader** es una envoltura para un flujo de bytes que maneja la lectura de datos binarios. Su constructor más utilizado se muestra a continuación:

```
BinaryReader(Stream flujoSalida)
```

Aquí, *flujoSalida* es el flujo del cual se leerán los datos. Para leer datos de salida a un archivo, puedes utilizar el objeto creado por **FileStream** para este parámetro. Si *flujoSalida* no ha sido abierto para datos de entrada o si es inválido de cualquier otra manera, se lanzará una excepción **ArgumentNullException**.

**BinaryReader** proporciona métodos para leer todos los tipos integrados de C#. Los de mayor uso se muestran en la tabla 11-6. **BinaryReader** también define tres versiones de **Read()**, que se muestran a continuación:

|                                            |                                                                                                                                                                                  |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| int Read()                                 | Regresa la representación en enteros del siguiente carácter disponible del flujo de entrada invocado. Regresa -1 cuando se intenta leer después del final de archivo.            |
| int Read(byte[ ] buf, int offset, int num) | Intenta leer hasta la cantidad de bytes determinada por <i>num</i> en <i>buf</i> , comenzando en <i>buf[offset]</i> , y regresa la cantidad de bytes leídos exitosamente.        |
| int Read(char[ ] buf, int offset, int num) | Intenta leer hasta la cantidad de caracteres determinada por <i>num</i> en <i>buf</i> , comenzando en <i>buf[offset]</i> , y regresa la cantidad de caracteres leídos con éxito. |

Estos métodos lanzarán una excepción **IOException** en caso de falla. También es posible que se generen otro tipo de excepciones. También define el método estándar **Close()**.

| Método                     | Descripción                                                    |
|----------------------------|----------------------------------------------------------------|
| bool ReadBoolean( )        | Lee un <b>bool</b> .                                           |
| byte ReadByte( )           | Lee un <b>byte</b> .                                           |
| sbyte ReadSByte( )         | Lee un <b>sbyte</b> .                                          |
| byte[ ] ReadBytes(int num) | Lee un <i>num</i> de bytes y los regresa como un arreglo.      |
| char ReadChar( )           | Lee un <b>char</b> .                                           |
| char[ ] ReadChars(int num) | Lee un <i>num</i> de caracteres y los regresa como un arreglo. |
| double ReadDouble( )       | Lee un <b>double</b> .                                         |
| float ReadSingle( )        | Lee un <b>float</b> .                                          |
| short ReadInt16( )         | Lee un <b>short</b> .                                          |
| int ReadInt32( )           | Lee un <b>int</b> .                                            |
| long ReadInt64( )          | Lee un <b>long</b> .                                           |
| ushort ReadUInt16( )       | Lee un <b>ushort</b> .                                         |
| uint ReadUInt32( )         | Lee un <b>uint</b> .                                           |
| ulong ReadUInt64( )        | Lee un <b>ulong</b> .                                          |
| string ReadString( )       | Lee una cadena de caracteres.                                  |

**Tabla 11-6** Métodos de entrada de mayor uso definidos por **BinaryReader**

## Mostrar E/S binaria

A continuación presentamos un programa que muestra el uso de **BinaryReader** y **BinaryWriter**. Escribe y luego lee varios tipos de datos desde y hacia un archivo.

```
// Escribe y luego lee datos binarios.
using System;
using System.IO;

class LeeEscribeData {
    static void Main() {
        BinaryWriter dataOut;
        BinaryReader dataIn;

        int i = 10;
        double d = 1023.56;
        bool b = true;

        try {
            dataOut = new
                BinaryWriter(new FileStream("testdata", FileMode.Create));
        }
```

```

        catch (IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }

        // Escribe datos en un archivo.
        try {
            Console.WriteLine("Escribiendo " + i);
            dataOut.Write(i); ←

            Console.WriteLine("Escribiendo " + d);
            dataOut.Write(d); ←

            Console.WriteLine("Escribiendo " + b);
            dataOut.Write(b); ←

            Console.WriteLine("Escribiendo " + 12.2 * 7.4);
            dataOut.Write(12.2 * 7.4); ←
        }
        catch (IOException exc) {
            Console.WriteLine(exc.Message);
        }

        dataOut.Close();

        Console.WriteLine();

        // Ahora, lee los datos.
        try {
            dataIn = new
                BinaryReader(new FileStream("testdata", FileMode.Open));
        }
        catch (IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }

        try {
            i = dataIn.ReadInt32(); ←
            Console.WriteLine("Leyendo " + i);

            d = dataIn.ReadDouble(); ←
            Console.WriteLine("Leyendo " + d); ←

            b = dataIn.ReadBoolean(); ←
            Console.WriteLine("Leyendo " + b);

            d = dataIn.ReadDouble(); ←
            Console.WriteLine("Leyendo " + d);
        }
    }
}

```

Escribe datos binarios.

Lee datos binarios.

(continúa)

```
        catch ( IOException exc ) {
            Console.WriteLine( exc.Message );
        }

        dataIn.Close();
    }
}
```

Los datos generados por el programa son los siguientes:

```
Escribiendo 10
Escribiendo 1023.56
Escribiendo True
Escribiendo 90.28

Leyendo 10
Leyendo 1023.56
Leyendo True
Leyendo 90.28
```

## Archivos de acceso aleatorio

Hasta este punto, hemos utilizado *archivos secuenciales*, que son archivos que son accesados de manera estrictamente lineal, un byte después de otro. Sin embargo, también puedes accesar el contenido de un archivo de manera aleatoria. Para hacerlo, utilizarás el método **Seek()** definido por **FileStream**. Este método te permite establecer el *indicador de posición de archivo* (también llamado *puntero de archivo*) en cualquier punto dentro del archivo.

El método **Seek()** se muestra a continuación:

```
long Seek(long nuevaPos, SeekOrigin origen)
```

Aquí, *nuevaPos* especifica la nueva posición, en bytes, del puntero de archivo a partir de la localización especificada por *origen*. El origen será uno de los siguientes valores, que son definidos por la enumeración **SeekOrigin**:

| Valor   | Significado                           |
|---------|---------------------------------------|
| Begin   | Busca desde el principio del archivo. |
| Current | Busca desde el punto actual.          |
| End     | Busca desde el final del archivo.     |

Después de una invocación a **Seek()**, la siguiente operación de lectura o escritura ocurrirá en la nueva posición del archivo. Si ocurre un error mientras se ejecuta la búsqueda, se lanzará una excepción **IOException**. Si el flujo subyacente no soporta requisiciones de posicionamiento, se lanzará una excepción **NotSupportedException**. También es posible que se generen otro tipo de excepciones.

He aquí un ejemplo que muestra el funcionamiento de acceso aleatorio de E/S. Escribe el alfabeto en mayúsculas en un archivo y luego lo lee en un orden aleatorio.

```

// Muestra el acceso aleatorio.
using System;
using System.IO;

class AccesoAleatorioDemo {
    static void Main() {
        FileStream f;
        char ch;

        try {
            f = new FileStream("random.dat", FileMode.Create);
        }
        catch (IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }

        // Escribe el alfabeto.
        for(int i=0; i < 26; i++) {
            try {
                f.WriteByte((byte)('A'+i));
            }
            catch (IOException exc) {
                Console.WriteLine(exc.Message);
                f.Close();
                return;
            }
        }

        try {
            // Ahora, lee valores específicos.
            f.Seek(0, SeekOrigin.Begin); // busca el primer byte ←
            ch = (char)f.ReadByte();
            Console.WriteLine("Primer valor es " + ch);

            f.Seek(1, SeekOrigin.Begin); // busca el segundo byte ←
            ch = (char)f.ReadByte();
            Console.WriteLine("Segundo valor es " + ch);

            f.Seek(4, SeekOrigin.Begin); // busca el quinto byte ←
            ch = (char)f.ReadByte();
            Console.WriteLine("Quinto valor es " + ch);

            Console.WriteLine();
        }

        // Ahora lee valores aleatoriamente.
        Console.WriteLine("Aquí están varios valores: ");
    }
}

```

**Utiliza `Seek()` para mover el puntero de archivo.**

```
        for(int i=0; i < 26; i += 2) {
            f.Seek(i, SeekOrigin.Begin); // busca el carácter i-ésimo
            ch = (char) f.ReadByte();
            Console.Write(ch + " ");
        }
    }
    catch (IOException exc) {
        Console.WriteLine(exc.Message);
    }

    Console.WriteLine();
    f.Close();
}
}
```

Los datos generados por el programa son:

```
Primer valor es A
Segundo valor es B
Quinto valor es E
```

```
Aquí están varios valores:
A C E G I K M O Q S U W Y
```

## Convertir cadenas numéricas en su representación interna

Antes de abandonar el tema de E/S, examinaremos una técnica de utilidad cuando se leen cadenas numéricas. Como sabes, **WriteLine()** proporciona una manera conveniente de mostrar diferentes tipos de datos a la consola, incluyendo valores numéricos de los tipos integrados, como **int** y **double**. Así, **WriteLine()** convierte automáticamente valores numéricos a su formato comprensible para el humano. Sin embargo, no existe un método de entrada paralelo que lea cadenas que contengan valores numéricos y los convierta a su formato binario interno. Por ejemplo, no hay una versión de **Read()** que lea del teclado una cadena como “100” y la convierta de manera automática a su correspondiente valor binario que pueda ser almacenado en una variable **int**. En lugar de ello, hay otros modos de realizar esta tarea. Tal vez, la manera más sencilla es utilizar un método que está definido para todos los tipos numéricos integrados: **Parse()**.

Antes de comenzar, es necesario dejar en claro un hecho importante: todos los tipos integrados C#, como **int** y **double**, son de hecho alias (esto es, sobrenombres) para estructuras definidas por .NET Framework. De hecho, el tipo C# y la estructura de tipos .NET son indistinguibles. Uno es sólo otro nombre para el otro. Dado que los tipos valor de C# son soportados por estructuras, los valores tipo tienen miembros definidos por ellos mismos.

Para los tipos de valor numérico de C#, los nombres de estructura .NET y su equivalente de tecla en C# son los siguientes:

| Nombre de estructura .NET | Nombre C# |
|---------------------------|-----------|
| Decimal                   | decimal   |
| Double                    | double    |
| Single                    | float     |
| Int16                     | short     |
| Int32                     | int       |
| Int64                     | long      |
| UInt16                    | ushort    |
| UInt32                    | uint      |
| UInt64                    | ulong     |
| Byte                      | byte      |
| SByte                     | sbyte     |

Estas estructuras están definidas dentro de la nomenclatura **System**. Así, el nombre completo calificado para **Int32** es **System.Int32**. Estas estructuras ofrecen una amplia variedad de métodos que ayudan a integrar completamente los tipos valor en la jerarquía de objetos de C#. Como beneficio secundario, las estructuras numéricas también definen métodos estáticos que convierten una cadena numérica en su correspondiente equivalente binario. Estos métodos de conversión se muestran a continuación. Cada uno regresa un valor binario que corresponde a la cadena.

| Estructura | Método de conversión             |
|------------|----------------------------------|
| Decimal    | static decimal Parse(string str) |
| Double     | static double Parse(string str)  |
| Single     | static float Parse(string str)   |
| Int64      | static long Parse(string str)    |
| Int32      | static int Parse(string str)     |
| Int16      | static short Parse(string str)   |
| UInt64     | static ulong Parse(string str)   |
| UInt32     | static uint Parse(string str)    |
| UInt16     | static ushort Parse(string str)  |
| Byte       | static byte Parse(string str)    |
| SByte      | static sbyte Parse(string str)   |

Los métodos **Parse( )** lanzarán una excepción **FormatException** si *str* no contiene un número válido definido por el tipo invocado. **ArgumentNullException** es lanzado si *str* es nula y **OverflowException** si el valor de *str* excede los límites del tipo invocado.

Los métodos de análisis (parseo) te ofrecen una manera sencilla de convertir un valor numérico, leído como cadena desde el teclado o de un archivo de texto, a su propio formato interno. Por ejemplo, el siguiente programa promedia una lista de números ingresados por el usuario. Primero pide al usuario la cantidad de valores que serán promediados. Luego lee ese número utilizando **ReadLine( )** y aplica **Int32.Parse( )** para convertir la cadena en un entero. A continuación, el programa ingresa los valores, utilizando **Double.Parse( )** para convertir la cadena en sus equivalentes **double**.

```
// Este programa promedia una lista de números ingresados por el usuario.
using System;
using System.IO;

class PromNums {
    static void Main() {
        string str;
        int n;
        double sum = 0.0;
        double avg, t;

        Console.WriteLine("Cuántos números ingresarás: ");
        str = Console.ReadLine();
        try {
            n = Int32.Parse(str); ←————— Convierte string en int.
            } catch (FormatException exc) {
                Console.WriteLine(exc.Message);
                return;
            } catch (OverflowException exc) {
                Console.WriteLine(exc.Message);
                return;
            }

            Console.WriteLine("Ingresa " + n + " valores.");
            for(int i=0; i < n; i++) {
                Console.Write(": ");
                str = Console.ReadLine();
                try {
                    t = Double.Parse(str); ←————— Convierte string a double.
                    } catch (FormatException exc) {
                        Console.WriteLine(exc.Message);
                        t = 0.0;
                    } catch (OverflowException exc) {
                        Console.WriteLine(exc.Message);
                        t = 0;
                    }
                    sum += t;
                }
            }
        }
    }
}
```

```
        }
        avg = sum / n;
        Console.WriteLine("El promedio es " + avg);
    }
}
```

Un ejemplo de los datos de salida que genera el programa:

```
Cuántos números ingresarás: 5
Ingresa 5 valores.
: 1.1
: 2.2
: 3.3
: 4.4
: 5.5
El promedio es 3.3
```

Puedes dar un buen uso a los métodos **Parse()** mejorando la calculadora de pagos desarrollada en el capítulo 2. En aquella versión, el préstamo inicial, intereses y demás, estaban “incrustados” en el programa. Sería más útil si el programa pidiera al usuario que ingresara tales datos. He aquí una versión de la calculadora de préstamos que lo hace:

```
// Calcula el pago regular para un préstamo, versión mejorada.

using System;

class PagoRegular {
    static void Main() {
        decimal Principal;      // préstamo original o principal
        decimal TasaInt;        // tasa de interés con decimales como 0.075
        decimal PagoAnual;       // cantidad de pagos por año
        decimal NumAños;         // número de años
        decimal Pago;            // el pago regular
        decimal numer, denom;   // variables de trabajo temporales
        double b, e;             // base y exponente para invocar Pow()

        string str;

        try {
            Console.Write("Ingrese préstamo original: ");
            str = Console.ReadLine();
            Principal = Decimal.Parse(str);

            Console.Write("Ingrese tasa de interés (como 0.085): ");
            str = Console.ReadLine();

            TasaInt = Decimal.Parse(str);
```

```
Console.Write("Ingrese la cantidad de años: ");
str = Console.ReadLine();
NumAños = Decimal.Parse(str);

Console.Write("Ingrese el número de pagos por año: ");
str = Console.ReadLine();
PagoAnual = Decimal.Parse(str);
} catch (FormatException exc) {
    Console.WriteLine(exc.Message);
    return;
} catch (OverflowException exc) {
    Console.WriteLine(exc.Message);
    return;
}

numer = TasaInt * Principal / PagoAnual;

e = (double)-(PagoAnual * NumAños);
b = (double)(TasaInt / PagoAnual) + 1;

denom = 1 - (decimal)Math.Pow(b, e);

Pago = numer / denom;

Console.WriteLine("El pago es {0:C}", Pago);
}
```

Un ejemplo de los datos que genera el programa:

```
Ingrese préstamo original: 10000
Ingrese tasa de interés (como 0.085): 0.075
Ingrese la cantidad de años: 5
Ingrese el número de pagos por año: 12
El pago es $200.38
```

## Pregunta al experto

**P:** ¿Qué más pueden hacer las estructuras numéricas tipo valor, como Int32 o Double?

**R:** Las estructuras numéricas tipo valor proporcionan una cantidad de métodos que ayudan a integrar los tipos integrados C# a la jerarquía de objetos. Por ejemplo, todas las estructuras tienen métodos llamados **CompareTo()**, que comparan los valores contenidos dentro de una envoltura; **Equals()**, que prueba dos valores para buscar su igualdad; y métodos que regresan el valor de los objetos en varios formatos. Las estructuras numéricas también incluyen los campos **MinValue** y **MaxValue**, que contienen los valores mínimo y máximo que pueden ser almacenados por un objeto de su tipo.

**Prueba esto**

## Crear un sistema de ayuda basado en disco

En la sección *Prueba esto* del capítulo 4, creaste una clase **Help** que muestra información sobre las declaraciones de control de C#. En esa implementación, la información de ayuda estaba almacenada dentro de la clase en sí, y el usuario hacía la selección de un menú con opciones numeradas. Aunque este enfoque era completamente funcional, no es de ninguna manera el medio ideal para crear un sistema de Ayuda. Por ejemplo, para añadir o cambiar la información de ayuda, se debe modificar el código fuente del programa. De la misma manera, seleccionar un tema por un número en lugar de hacerlo por su nombre resulta tedioso e inconveniente para listas largas de elementos. Aquí remediamos estas desventajas creando un sistema de ayuda basado en disco.

El sistema de ayuda basado en disco almacena la información de ayuda en un archivo. Este archivo de ayuda es un archivo de texto estándar, que puede ser modificado o expandido a voluntad, sin necesidad de modificar el programa de ayuda. El usuario obtiene el tema de ayuda tecleando su nombre. El sistema de ayuda busca el tema requerido en el archivo de texto. En caso de encontrarlo, se muestra la información del tema.

## Paso a paso

1. Debes crear el archivo de texto que será utilizado por el sistema de ayuda. Se trata de un archivo de texto estándar que se organiza de la siguiente manera:

```
#nombre-tema1  
información sobre el tema  
  
#nombre-tema2  
información sobre el tema  
  
. . .  
  
#nombre-temaN  
información sobre el tema
```

El nombre de cada tema debe ir antecedido por un #, y cada nombre debe ocupar su propia línea. Colocar el # al principio de cada nombre permite que el programa identifique rápidamente su inicio. Después del nombre puede ir un número o cualquier otra información sobre el mismo. Sin embargo, debe existir una línea en blanco entre el final de la información de un tema y el principio del siguiente nombre. Además, *no deben existir espacios en blanco* al final de ninguna línea.

A continuación presentamos un archivo sencillo que puedes utilizar para probar el sistema de ayuda basado en disco. Contiene información sobre las declaraciones de control de C#.

(continúa)

```
#if
if(condición) declaración;
else declaración;
#switch
switch(expresión) {
    case constante:
        secuencia de declaraciones
        break;
        // ...
}

#for
for(init; condición; reiteración) declaración;

}while
while(condición) declaración;

#do
do {
    declaración;
} while (condición);

#break
break; or break etiqueta;

#continue
continue; o continue etiqueta;

#goto
goto etiqueta;
```

Guarda este archivo con el nombre **helpfile.txt** (archivo de ayuda).

- 2.** Crea un archivo llamado **ArchivoAyuda.cs**.
- 3.** Comienza creando la nueva clase **Ayuda** con estas líneas de código:

```
class Ayuda {
    string helpfile; // nombre del archivo de ayuda

    public Ayuda(string fname) {
        helpfile = fname;
    }
}
```

El nombre del archivo de ayuda es transmitido al constructor **Ayuda** y se almacena en la variable de instancia **helpfile**. Como cada instancia de **Ayuda** tendrá su propia copia de **helpfile**, cada instancia puede utilizar un archivo distinto. De esta manera, puedes crear diferentes conjuntos de archivos de ayuda para distintos conjuntos de temas.

- 4.** Añade el método **HelpOn()** que se muestra a continuación a la clase **Ayuda**. Este método recupera la información de ayuda sobre el tema especificado.

```
// Muestra ayuda sobre el tema.
public bool HelpOn(string what) {
    StreamReader helpRdr;
    int ch;
    string topic, info;

    try {
        helpRdr = new StreamReader(helpfile);
    }
    catch (IOException exc) {
        Console.WriteLine(exc.Message);
        return false;
    }

    try {
        do {
            // Lee los caracteres hasta que localiza un #.
            ch = helpRdr.Read();

            // Ahora, comprueba que el tema coincida.
            if (ch == '#') {
                topic = helpRdr.ReadLine();
                if (what == topic) { // encuentra tema
                    do {
                        info = helpRdr.ReadLine();
                        if (info != null) Console.WriteLine(info);
                    } while ((info != null) && (info != ""));
                    helpRdr.Close();
                    return true;
                }
            }
        } while (ch != -1);
    }
    catch (IOException exc) {
        Console.WriteLine(exc.Message);
    }
    helpRdr.Close();
    return false; // tema no encontrado
}
```

El archivo de ayuda se abre utilizando **StreamReader**. Como este archivo contiene texto, utilizar un flujo de caracteres permite que el sistema de Ayuda sea más eficientemente internacionalizado.

El método **HelpOn( )** funciona así: la cadena que contiene el nombre del tema es transmitida en el parámetro **what**. Entonces se abre el archivo de ayuda. A continuación, se busca en el archivo por una coincidencia entre **what** y un tema dentro del archivo. Recuerda: en el archivo, cada tema es antecedido por un #, así que el loop busca estos signos en el archivo. Cuando encuentra uno, verifica si el tema que sigue al # coincide con el que se transmitió en **what**. De ser así, se muestra la información asociada con ese tema. Si se localiza una coincidencia, el método **HelpOn( )** regresa **true**. En caso contrario, regresa **false**.

(continúa)

- 5.** La clase Ayuda también proporciona un método llamado **GetSelection()**, que se muestra a continuación. Pide al usuario que escriba un tema y regresa la cadena del tema proporcionada por el usuario.

```
// Obtiene un tema de ayuda.
public string GetSelection() {
    string topic = "";

    Console.Write("Escribe el tema: ");
    topic = Console.ReadLine();
    return topic;
}
```

- 6.** A continuación se muestra el programa completo del sistema de Ayuda basado en disco:

```
// Un programa de ayuda que utiliza un archivo de disco para almacenar
// la información de ayuda.

using System;
using System.IO;

/* La clase Ayuda abre un archivo de ayuda},
   busca un tema y luego muestra
   la información asociada con ese tema. */
class Ayuda {
    string helpfile; // nombre del archivo de ayuda

    public Ayuda(string fname) {
        helpfile = fname;
    }

    // Muestra ayuda sobre el tema.
    public bool HelpOn(string what) {
        StreamReader helpRdr;
        int ch;
        string topic, info;

        try {
            helpRdr = new StreamReader(helpfile);
        }
        catch (IOException exc) {
            Console.WriteLine(exc.Message);
            return false;
        }

        try {
            do {
                // Lee los caracteres hasta que localiza un #.
                ch = helpRdr.Read();

                // Ahora, comprueba que el tema coincida.
                if(ch == '#') {

```

```
topic = helpRdr.ReadLine();
if(what == topic) { // encuentra tema
    do {
        info = helpRdr.ReadLine();
        if(info != null) Console.WriteLine(info);
    } while ((info != null) && (info != ""));
    helpRdr.Close();
    return true;
}
}
} while (ch != -1);
}
catch (IOException exc) {
    Console.WriteLine(exc.Message);
}
helpRdr.Close();
return false; // tema no encontrado
}

// Obtiene un tema de ayuda.
public string GetSelection() {
    string topic = "";

    Console.Write("Escribe el tema: ");
    topic = Console.ReadLine();
    return topic;
}

// Muestra el funcionamiento del sistema de Ayuda basado en disco.
class ArchivoAyuda {
    static void Main() {
        Ayuda hlpobj = new Ayuda("helpfile.txt");
        string topic;

        Console.WriteLine("Prueba el sistema de ayuda. " +
                          "Escribe 'stop' para finalizar.");
        for( ; ; ) {
            topic = hlpobj.GetSelection();

            if(topic == "stop") break;

            if(!hlpobj.HelpOn(topic))
                Console.WriteLine("Tema no encontrado.\n");
        }
    }
}
```

## Pregunta al experto

**P:** Antes, en este capítulo, mencionó la clase de flujo **MemoryStream**, que utiliza memoria para el almacenamiento. ¿Cómo se puede utilizar este flujo?

**R:** **MemoryStream** es una implementación de **Stream** que utiliza un arreglo de bytes para datos de entrada o de salida. He aquí uno de los constructores que lo define:

`MemoryStream(byte[ ]buf)`

Aquí, *buf* es un arreglo de bytes que será utilizado por la fuente o el destino de la requisición E/S. Debe ser lo bastante grande para guardar cualquier dato de salida que le dirijas. El flujo creado por este constructor puede ser de lectura o escritura y soporta **Seek()**.

Los flujos basados en memoria son muy útiles en programación. Por ejemplo, puedes construir salidas complejas por adelantado y almacenarlas en el arreglo hasta que se necesiten. Esta técnica es especialmente útil cuando se programa para ambientes gráficos, como Windows. También puedes redireccionar un flujo estándar para que lea del arreglo. Esto puede ser útil para proporcionar información de prueba en un programa, por ejemplo.

Un último punto: para crear un flujo de caracteres basado en memoria, utiliza **StringReader** o **StringWriter**.



## Autoexamen Capítulo 11

1. ¿Por qué C# define flujos tanto de byte como de carácter?
2. ¿Qué clase está al tope de la jerarquía de flujo?
3. Muestra cómo abrir un archivo para leer bytes.
4. Muestra cómo abrir un archivo para leer caracteres.
5. ¿Qué acciones realiza **Seek()**?
6. ¿Qué clases soportan E/S binaria para los tipos integrados?
7. ¿Qué métodos se utilizan para redireccionar los flujos estándar bajo el control del programa?
8. ¿Cómo conviertes una cadena numérica como “123.23” a su equivalente binario?
9. Escribe un programa que copie un archivo de texto. En el proceso, haz que convierta todos los espacios en guiones. Utiliza las clases de archivo de flujo de bytes.
10. Reescribe el programa de la pregunta 9 para que utilice las clases de flujo de caracteres.

# Capítulo 12

Delegados, eventos  
y nomenclaturas

## Habilidades y conceptos clave

- Delegados
  - Utilizar métodos de instancia con delegados
  - Delegados de distribución múltiple
  - Métodos anónimos
  - Eventos
  - Uso de métodos anónimos con eventos
  - Eventos de distribución múltiple
  - Nomenclaturas
  - La directiva **using**
  - Añadir nomenclaturas
  - Nomenclaturas anidadas
- 

Los pasados once capítulos han introducido lo que puede describirse como el corazón de C#. Cubren los temas fundamentales, como tipos de datos, declaraciones de control, clases, métodos y objetos. También abordaron varios conceptos clave como herencia, interfaces, estructuras, indexadores, propiedades, sobrecargas, excepciones y E/S. En este punto de tu estudio, ya puedes empezar a escribir programas útiles. Sin embargo, C# ofrece mucho más. El resto de este libro presenta algunas de las características más sofisticadas o (en algunos casos) más avanzadas de C#. Éste es un buen momento para detenerse y hacer una revisión. Querrás tener una base bien establecida antes de continuar.

Este capítulo examina tres importantes temas de C#: delegados, eventos y nomenclaturas. Los delegados son, esencialmente, objetos que pueden hacer referencia a código ejecutable. Los eventos se construyen sobre delegados. Un evento es un aviso de que ocurrió alguna acción. Las nomenclaturas ayudan a organizar tu código. Juntas, estas características añaden mucho al poder expresivo de C# y se utilizan ampliamente en el código comercial.

### **NOTA**

El presente y los siguientes capítulos de este libro presentan una introducción a las características más sofisticadas y poderosas de la programación en C#. Muchas de ellas soportan opciones y técnicas que rebasan los objetivos de esta guía para principiantes. Mientras progresas en tu conocimiento sobre C#, querrás aprender más acerca de estas características. Un buen punto de partida es mi libro *C# 3.0: Manual de referencia*.

## Delegados

Los novatos en C# algunas veces se sienten intimidados por los delegados, pero no hay nada que temer. Los delegados no son más difíciles de entender o utilizar que cualquier otra característica de C#, mientras tengas en mente, con toda precisión, lo que es un delegado. En términos sencillos, un *delegado* es un objeto que puede hacer referencia a un método. Así, cuando creas un delegado, estás creando un objeto que puede almacenar una referencia a un método. Más aún, el método puede de ser invocado a través de esa referencia. Así, un delegado puede invocar el método al cual hace referencia.

Una vez que un delegado hace referencia a un método, éste puede ser invocado a través del delegado. Además, el mismo delegado puede utilizarse para invocar diferentes métodos, simplemente cambiando el método al que hace referencia. La principal ventaja de un delegado es que te permite especificar la invocación a un método, pero el método que de hecho se invoca se determina en tiempo de ejecución y no en tiempo de compilación.

### **NOTA**

Si estás familiarizado con C/C++, será de utilidad saber que un delegado en C# es similar al puntero de función en C/C++.

Un tipo delegado se declara utilizando la palabra clave **delegate**. El formato general de una declaración delegado es el siguiente:

`delegate tipo-ret nombre(lista-parámetros);`

Aquí, *tipo-ret* es el tipo de valor regresado por los métodos que el delegado invocará. El nombre del delegado se especifica en *nombre*. Los parámetros requeridos por los métodos invocados a través del delegado se especifican en la *lista-parámetros*. Una vez creada, una instancia delegado puede hacer referencia e invocar sólo métodos cuyo tipo y lista de parámetros coincidan con aquellos especificados por la declaración del delegado.

Un punto clave que debe comprenderse es que un delegado puede utilizarse para invocar *cualquier* método que coincida con su firma y tipo de regreso. Más aún, el método puede especificarse en tiempo de ejecución simplemente asignando al delegado una referencia a un método compatible. El método invocado puede ser uno de instancia asociado con un objeto o un método estático asociado con una clase. Todo lo que importa es que la firma y el tipo de regreso del método coincidan con los del delegado.

Para ver los delegados en acción, comenzemos con el ejemplo sencillo que se muestra a continuación:

```
// Un ejemplo sencillo de delegado.
using System;

// Declara un delegado.
delegate string StrMod(string str); ←———— Un delegado llamado StrMod.

class DelegadoTest {
    // Reemplaza espacios con guiones.
    static string ReemplazaEspacios(string a) {
```

## 434 Fundamentos de C# 3.0

---

```
Console.WriteLine("Reemplaza espacios con guiones.");
    return a.Replace(' ', '-');
}

// Elimina espacios.
static string EliminaEspacios(string a) {
    string temp = "";
    int i;

    Console.WriteLine("Elimina espacios.");
    for(i=0; i < a.Length; i++)
        if(a[i] != ' ') temp += a[i];

    return temp;
}

// Invierte una cadena.
static string Invierte(string a) {
    string temp = "";
    int i, j;

    Console.WriteLine("Invierte una cadena.");
    for(j=0, i=a.Length-1; i >= 0; i--, j++)
        temp += a[i];

    return temp;
}

static void Main() {
    // Construye un delegado.
    StrMod strOp = ReemplazaEspacios; ← Construye una instancia delegado.
    string str;

    // Invoca métodos a través del delegado.
    str = strOp("Ésta es una prueba."); ← Invoca un método desde delegado.
    Console.WriteLine("Cadena resultante: " + str);
    Console.WriteLine();

    strOp = EliminaEspacios;
    str = strOp("Ésta es una prueba.");
    Console.WriteLine("Cadena resultante: " + str);
    Console.WriteLine();

    strOp = Invierte;
    str = strOp("Ésta es una prueba.");
    Console.WriteLine("Cadena resultante: " + str);
}
}
```

Los datos generados por el programa son los siguientes:

```
Reemplaza espacios con guiones.  
Cadena resultante: Ésta-es-una-prueba.
```

```
Elimina espacios.  
Cadena resultante: Éstaesunaprueba.
```

```
Invierte una cadena.  
Cadena resultante: .abeurp anu se atsÉ
```

Examinemos este programa de cerca. El programa declara un delegado llamado **StrMod** que toma un parámetro **string** y regresa un **string**. En **DelegadoTest** se declaran tres métodos estáticos, cada uno con una firma coincidente. Estos métodos realizan cierta modificación al texto. Observa que **ReemplazaEspacios()** utiliza uno de los métodos de **string**, llamado **Replace()**, para reemplazar los espacios en blanco por guiones.

En **Main()**, una referencia **StrMod** llamada **strOp** es creada y una referencia es asignada a **ReplaceSpaces()**. Pon especial atención a esta línea:

```
StrMod strOp = ReemplazaEspacios;
```

Observa cómo el método **ReemplazaEspacios()** es asignado a **strOp**. Sólo se utiliza su nombre, no se especifica ningún parámetro. Esta sintaxis puede generalizarse a cualquier situación en la cual un método es asignado a un delegado. Esta sintaxis funciona porque C# proporciona automáticamente una conversión de método a tipo delegado. La acción recibe el nombre de *conversión de grupo de método* y es una característica añadida desde la versión 2.0 de C#. En realidad se trata de la abreviación de un formato más largo:

```
strMod strOp = new StrMod(ReemplazaEspacios);
```

En este formato se inicializa explícitamente un nuevo delegado utilizando **new**. Aunque todavía encontrarás el formato largo en uso dentro de algún código, el formato anterior es más sencillo y se utiliza más. Con cualquier formato, la firma del método debe coincidir con la declaración del delegado. De no ser así, dará como resultado un error en tiempo de compilación.

A continuación, **ReemplazaEspacios()** es invocado a través de la instancia delegado **strOp**, como se muestra aquí:

```
str = strOp("Esto es una prueba.");
```

Como **strOp** hace referencia a **ReemplazaEspacios()**, es este último el que se invoca.

Paso seguido, a **strOp** se le asigna una referencia a **EliminaEspacios()**, y luego **strOp** es invocada de nuevo. Esta vez, se invoca **EliminaEspacios()**.

Finalmente, a **strOp** se le asigna una referencia a **Invierte()** y se invoca **strOp**. Esto da como resultado la invocación de **Invierte()**.

El punto clave del ejemplo es que la invocación a **strOp** da como resultado una invocación del método al que hace referencia **strOp** en el momento en que ocurre la invocación. De esta manera, el método a invocar se resuelve en tiempo de ejecución, y no durante la compilación.

## Utilizar métodos de instancia como delegados

Aunque el ejemplo anterior utiliza métodos estáticos, un delegado también puede hacer referencia a métodos de instancia. Sin embargo, debe hacerlo a través de una referencia de objeto. Por ejemplo, a continuación presentamos una versión diferente del programa anterior, que encapsula las operaciones de cadena dentro de una clase llamada **StringOps**:

```
// El delegado puede hacer referencia a métodos de instancia también.
using System;

// Declara un tipo delegado.
delegate string StrMod(string str);

class StringOps {
    // Reemplaza espacios con guiones.
    public string ReemplazaEspacios(string a) {
        Console.WriteLine("Reemplaza espacios con guiones.");
        return a.Replace( ' ', '-' );
    }

    // Elimina espacios.
    public string EliminaEspacios(string a) {
        string temp = "";
        int i;

        Console.WriteLine("Elimina espacios.");
        for(i=0; i < a.Length; i++)
            if(a[i] != ' ') temp += a[i];

        return temp;
    }

    // Invierte una cadena.
    public string Invierte(string a) {
        string temp = "";
        int i, j;

        Console.WriteLine("Invierte una cadena.");
        for(j=0, i=a.Length-1; i >= 0; i--, j++)
            temp += a[i];

        return temp;
    }
}

class DelegadoTest {
    static void Main() {
        StringOps so = new StringOps();
```

```

// Construye un delegado.
StrMod strOp = so.ReemplazaEspacios; ← Crea un delegado usando
string str; un método de instancia.

// Invoca métodos desde un delegado.
str = strOp("Ésta es una prueba.");
Console.WriteLine("Cadena resultante: " + str);
Console.WriteLine();

strOp = so.EliminaEspacios;
str = strOp("Ésta es una prueba.");
Console.WriteLine("Cadena resultante: " + str);
Console.WriteLine();

strOp = so.Invierte;
str = strOp("Ésta es una prueba.");
Console.WriteLine("Cadena resultante: " + str);
}

}

```

Esta versión del programa genera los mismos resultados que la anterior, sólo que en este caso el delegado hace referencia a métodos en una instancia de **StringOps**.

## Distribución múltiple

Una de las características más excitantes de los delegados es que soportan distribución múltiple (*multicasting*). En términos sencillos, la distribución múltiple es la capacidad de crear una cadena de métodos que serán invocados automáticamente cuando el delegado sea invocado. Esa cadena es muy sencilla de crear. Simplemente inicializa un delegado y luego usa el operador `+` o el `+≡` para añadir métodos a la cadena. Para eliminar un método utiliza `-` o `-≡`. Si el delegado regresa un valor, entonces el valor regresado por el último método en la lista se convierte en el valor de regreso de la invocación completa al delegado. Por esta razón, un delegado que hace uso de la distribución múltiple por lo regular tiene un tipo de retorno **void**.

He aquí un ejemplo de distribución múltiple. Reelabora los ejemplos anteriores al cambiar a **void** el tipo de regreso del método que manipula la cadena de caracteres y utiliza un parámetro **ref** para regresar la cadena alterada al invocador.

```

// Muestra la distribución múltiple.
using System;

// Declara un tipo delegado.
delegate void StrMod(ref string str);

class StringOps {
    // Reemplaza los espacios por guiones.
    static void ReemplazaEspacios(ref string a) {
        Console.WriteLine("Reemplaza los espacios por guiones.");
        a = a.Replace(' ', '-');
    }
}

```

```
// Elimina espacios.  
static void EliminaEspacios(ref string a) {  
    string temp = "";  
    int i;  
  
    Console.WriteLine("Elimina espacios.");  
    for(i=0; i < a.Length; i++)  
        if(a[i] != ' ') temp += a[i];  
  
    a = temp;  
}  
  
// Invierte una cadena.  
static void Invierte(ref string a) {  
    string temp = "";  
    int i, j;  
  
    Console.WriteLine("Invierte una cadena.");  
    for(j=0, i=a.Length-1; i >= 0; i--, j++)  
        temp += a[i];  
  
    a = temp;  
}  
  
static void Main() {  
    // Construye un delegado.  
    StrMod strOp;  
    StrMod reemplazaSp = ReemplazaEspacios;  
    StrMod eliminaSp = EliminaEspacios;  
    StrMod invierteStr = Invierte;  
    string str = "Ésta es una prueba";  
  
    // Establece la distribución múltiple.  
    strOp = reemplazaSp;  
    strOp += invierteStr; ←———— Crea una distribución múltiple.  
  
    // Invoca la distribución múltiple.  
    strOp(ref str);  
    Console.WriteLine("Cadena resultante: " + str);  
    Console.WriteLine();  
  
    // Elimina replaceSp y añade removeSp.  
    strOp -= reemplazaSp; ←———— Crea una distribución múltiple diferente.  
    strOp += eliminaSp;  
  
    str = "Ésta es una prueba."; // restablece la cadena  
    // Invoca la distribución múltiple.  
    strOp(ref str);
```

```

        Console.WriteLine("Cadena resultante: " + str);
        Console.WriteLine();
    }
}

```

Los datos de salida generados por el programa:

```

Reemplaza los espacios por guiones.
Invierte una cadena.
Cadena resultante: abeurn-anu-se-atsÉ

```

```

Invierte una cadena.
Elimina espacios.
Cadena resultante:.abeurpanuseatsÉ

```

En **Main()** se crean cuatro instancias delegado. Una, **strOp**, es nula. Las otras tres hacen referencia a métodos específicos que modifican cadenas. A continuación se crea una distribución múltiple que invoca **ReemplazaEspacios()** e **Invierte()**. Esto se realiza con las siguientes líneas:

```

strOp = reemplazaSp;
strOp += invierteStr;

```

Primero, a **strOp** se le asigna una referencia a **reemplazaSp**. A continuación, utilizando **+=**, se añade **invierteStr**. Cuando **strOp** se invoca, ambos métodos son invocados, reemplazando espacios con guiones e invirtiendo la cadena, como lo muestran los datos de salida.

Paso siguiente, **reemplazaSp** se elimina de la cadena de métodos utilizando esta línea:

```
strOp -= reemplazaSp;
```

y se añade **eliminaSp** utilizando esta línea:

```
strOp += eliminaSp;
```

Entonces, **strOp** se vuelve a invocar. Esta vez, los espacios son eliminados y la cadena de caracteres se invierte.

## Por qué delegados

Aunque los anteriores ejemplos muestran el “cómo” detrás de los delegados, en realidad no ilustran el “porqué”. En general, los delegados son útiles por dos razones principales. La primera, como pronto verás, los delegados soportan eventos. La segunda, los delegados le dan a tu programa un medio para ejecutar métodos en tiempo de ejecución sin necesidad de especificar de qué método se trata en tiempo de compilación. Esta capacidad es muy útil cuando quieras crear una estructura que permita conectar componentes. Por ejemplo, imagina un programa de dibujo (un poco parecido al estándar del accesorio Windows Paint). Al utilizar un delegado le puedes ofrecer al usuario la capacidad de conectar filtros de color especiales o analizadores de imágenes. Más aún, el usuario podría crear una secuencia de estos filtros o analizadores. Un esquema así sería fácil de manejar utilizando delegados.

## Métodos anónimos

Cuando se trabaja con delegados, con frecuencia encontrarás que el método al que hace referencia el delegado se utiliza únicamente con ese propósito. En otras palabras, la única razón de ser del método es que sea invocado por el delegado. Este método nunca es invocado por sí mismo. En tales casos, puedes evitar la necesidad de crear un método separado utilizando un *método anónimo*. Un método anónimo es, en esencia, un bloque de código que es transmitido al constructor del delegado. Una ventaja de utilizar métodos anónimos es la sencillez. No hay necesidad de declarar un método separado cuyo único propósito es ser transmitido a un delegado.

Antes de continuar, es necesario aclarar un punto de importancia. C# 3.0 añade una nueva característica, la expresión lambda, la cual (en muchos casos) mejora el concepto del método anónimo. Las expresiones lambda serán abordadas en el capítulo 14. Aunque las opciones lambda son por lo regular una mejor opción, no se aplican a todas las situaciones. Además, los métodos anónimos son muy utilizados en el código actualmente existente de C#. Por lo mismo, es importante que comprendas su funcionamiento y que las veas en acción.

He aquí un ejemplo que utiliza un método anónimo:

```
// Muestra un método anónimo en acción.
using System;

// Declara un delegado.
delegate void Cuenta();

class AnonMethDemo {

    static void Main() {

        // Aquí, el código para contar se transmite
        // como un método anónimo.
        Cuenta count = delegate {
            // Éste es el bloque de código que se transmite al delegado.
            for(int i=0; i <= 5; i++)
                Console.Write(i + " ");
        }; // observa el punto y coma

        count();
        Console.WriteLine();
    }
}
```



Un método anónimo.

Los datos generados por el programa son:

0 1 2 3 4 5

Este programa primero declara un tipo delegado llamado **Cuenta** que no tiene parámetros y su regreso es **void**. Dentro de **Main()**, se crea un delegado **Cuenta** llamado **count** y se le transmite el bloque de código que sigue a la palabra clave **delegate**. Este bloque de código es el método anóni-

mo que se ejecutará cuando **count** sea invocado. Observa que el bloque de código es seguido por un punto y coma (;), que finaliza la declaración.

Es posible transmitir uno o más argumentos a un método anónimo. Para hacerlo, después de la palabra clave **delegate** añade un juego de paréntesis con una lista de parámetros. Luego, transmite el o los argumentos a la instancia delegado cuando se invoque. Por ejemplo, a continuación presentamos una versión nueva del ejemplo anterior donde se transmite el valor final de la cuenta:

```
// Muestra un método anónimo que recibe un argumento.  
using System;  
  
// Observa que Cuenta no tiene parámetros.  
delegate void Cuenta(int end);  
  
class AnonMethDemo2 {  
  
    static void Main() {  
  
        // Aquí, el valor final de la cuenta  
        // es transmitido al método anónimo.  
        Cuenta count = delegate(int end) { ← Utiliza un parámetro con  
            for(int i=0; i <= end; i++)  
                Console.WriteLine(i + " ");  
        };  
  
        count(3);  
        Console.WriteLine();  
        count(5);  
        Console.WriteLine();  
    }  
}
```

En esta versión, **Cuenta** toma un número entero como argumento. Observa cómo la lista de parámetros se especifica después de la palabra clave **delegate** cuando se crea el método anónimo. En este caso, el único parámetro es **end**, que en realidad es un **int**. El código dentro del método anónimo tiene acceso al parámetro **end** de la misma manera en que lo tendría si se estuviera creando un método “normal”. Los datos de salida que genera el programa son los siguientes:

```
0 1 2 3  
0 1 2 3 4 5
```

Como puedes ver, el valor del argumento para **count()** es recibido por **end**, y este valor es utilizado como el punto final de la cuenta.

Un método anónimo puede regresar un valor. El valor es regresado utilizando la declaración **return**, que funciona de la misma manera en un método anónimo que en un método con nombre. Como podrías esperar, el tipo del valor de regreso debe ser compatible con el tipo de regreso especificado por el delegado.

El siguiente ejemplo muestra un método anónimo que regresa un valor. Cambia el código de cuenta de manera que regrese una sumatoria de los números.

```
// Muestra un método anónimo que regresa un valor.
using System;

// Este delegado regresa un valor.
delegate int Cuenta(int end);

class AnonMethDemo3 {

    static void Main() {
        int result;

        // Aquí el valor final de la cuenta
        // es transmitido al método anónimo.
        // Regresa una sumatoria de la cuenta.
        Cuenta count = delegate(int end) {
            int sum = 0;

            for(int i=0; i <= end; i++) {
                Console.Write(i + " ");
                sum += i;
            }
            return sum; // regresa un valor de un método anónimo
        };
    }

    result = count(3);
    Console.WriteLine("\nSumatoria de 3 es " + result);
    Console.WriteLine();

    result = count(5);
    Console.WriteLine("\nSumatoria de 5 es " + result);
    Console.WriteLine();
}

}
```

↑  
Un método anónimo puede regresar un valor.

En esta versión, el valor de **sum** es regresado por el bloque de código que está asociado con la instancia delegado **count**. Observa que la declaración de regreso es utilizada en el método anónimo de la misma manera como se utiliza en un método “normal”. Los datos generados por el programa son:

```
0 1 2 3
Sumatoria de 3 es 6

0 1 2 3 4 5
Sumatoria de 5 es 15
```

Tal vez el uso más importante de un método anónimo es con eventos. Como verás más adelante en este mismo capítulo, en muchas ocasiones los métodos anónimos son el medio más eficiente de codificar un controlador de eventos.

## Pregunta al experto

**P:** He escuchado el término *variables externas* utilizado en discusiones sobre los métodos anónimos. ¿A qué se refiere?

**R:** Una variable local o un parámetro cuyo alcance incluye un método anónimo es llamada *variable externa*. Un método anónimo tiene acceso y puede utilizar variables externas. Cuando una variable externa es utilizada por un método anónimo, se dice que la variable está *capturada*. Una variable capturada permanecerá en existencia hasta que el método que la captura sea sujeto a la recolección de basura. De esta manera, aunque una variable local por lo general deja de existir cuando el flujo del programa abandona su bloque, si esa variable local es utilizada por un método anónimo, entonces dicha variable permanecerá en existencia por lo menos hasta que el delegado que hace referencia a ese método sea destruido.

## Eventos

Otra característica importante de C# son los *eventos*. Un evento es, en esencia, la notificación automática de que una acción ha ocurrido. Los eventos son muy utilizados en el código aplicable al mundo real porque se les utiliza para representar cosas como los golpes de tecla, clics del ratón, requisiciones para refrescar la pantalla y los datos entrantes. Los eventos se construyen sobre la base de los delegados. Así, necesitas entender los delegados para poder utilizar los eventos.

Los eventos funcionan así: un objeto que tiene interés en un evento registra un controlador de eventos para éste. Cuando el evento ocurre, todos los controladores registrados son invocados. Los controladores de eventos están representados por delegados. El controlador de eventos responde al evento tomando las acciones apropiadas. Por ejemplo, un controlador de eventos para los golpes de teclas puede responder enviando el carácter asociado con la tecla al monitor. Como regla general, un evento debe responder rápidamente y después regresar. No debe mantener control del CPU durante mucho tiempo. Como los eventos son utilizados normalmente para manejar actividades en tiempo real, un evento que domine el CPU puede impactar de manera negativa el rendimiento general y las respuestas del programa.

Los eventos son miembros de una clase y se declaran utilizando la palabra clave **event**. Su formato general se muestra a continuación:

*event delegado-evento nombre-evento;*

Aquí, *delegado-evento* es el nombre del delegado que se utiliza para soportar el evento, y *evento-nombre* es el nombre del evento específico que se está declarando.

Comencemos con un ejemplo muy sencillo:

```
// Una demostración muy sencilla de evento.  
using System;  
  
// Declara un tipo delegado para un evento.  
delegate void MiControladorEventos(); ← Crea un delegado para el evento.  
  
// Declara una clase que contiene un evento.  
class MiEvento {  
    public event MiControladorEventos UnEvento; ← Declara un evento.  
  
    // Esto se invoca para disparar el evento.  
    public void Dispara() {  
        if (UnEvento != null)  
            UnEvento(); ← Dispara el evento.  
    }  
}  
  
class EventoDemo {  
    static void Contolador() { ← Un controlador de eventos.  
        Console.WriteLine("Ocurrió un evento");  
    }  
  
    static void Main() {  
        MiEvento evt = new MiEvento(); ← Crea una instancia de evento.  
  
        // Añade Controlador() a la lista de eventos.  
        evt.UnEvento += Contolador; ← Añade un controlador a la cadena de eventos.  
  
        // Dispara evento.  
        evt.Dispara(); ← Genera el evento.  
    }  
}
```

El programa muestra los siguientes datos de salida:

```
Ocurrió un evento
```

Aunque sencillo, este programa contiene todos los elementos esenciales para controlar adecuadamente un evento. Veámoslo con detalle.

El programa inicia declarando un tipo delegado para el controlador de eventos, como se muestra aquí:

```
delegate void MiControladorEventos();
```

Todos los eventos se activan a través de un delegado. Así, el tipo delegado de eventos define el tipo de regreso y la firma para el evento. En este caso no hay parámetros, pero sí están permitidos los parámetros para eventos.

A continuación, se crea una clase evento, llamada **MiEvento**. Dentro de esa clase se declara el evento llamado **UnEvento**, utilizando esta línea:

```
public event MiControladorEventos UnEvento;
```

Observa la sintaxis. La palabra clave **event** le indica al compilador que ha sido declarado un evento.

También dentro de **MiEvento** está declarado el método **Dispara()**, que es el método que invocará el programa para señalar (o disparar) un evento. Invoca un controlador de eventos a través del delegado **UnEvento**, como se muestra aquí:

```
if(UnEvento != null)
    UnEvento();
```

Observa que el controlador es invocado si y sólo si **UnEvento** no es **null**. Como otras partes de tu programa deben registrar un interés en un evento con el fin de recibir notificaciones sobre el mismo, es posible que **Dispara()** pueda ser invocado antes de que cualquier controlador de eventos haya sido registrado. Para prevenir la invocación a una referencia **null**, el delegado del evento debe ser probado para asegurarse de que no es **null**.

Dentro de **EventoDemo**, se crea un controlador de eventos llamado **Controlador()**. En este ejemplo sencillo, el controlador de eventos simplemente muestra un mensaje, pero por lo regular los controladores realizan acciones más significativas. En **Main()**, se crea un objeto **MiEvento**, y **Controlador()** es registrado como el controlador de ese evento, como se muestra aquí:

```
MiEvento evt = new MiEvento();
// Añade Controlador() a la lista de eventos.
evt.UnEvento += Controlador;
```

Observa que el controlador se añade utilizando el operador **+=**. Los eventos sólo soportan **+=** y **-=** para añadir o eliminar controladores.

Finalmente, el evento se dispara, como se muestra a continuación:

```
// Dispara evento.
evt.Dispara();
```

Invocar **Dispara()** hace que se invoquen todos los controladores de registro. En este caso, sólo hay uno registrado, pero podría haber más, como se explica en la siguiente sección.

## Un ejemplo de evento de distribución múltiple

La distribución múltiple se puede aplicar a los eventos. Esto permite que múltiples objetos respondan a la notificación de un evento. He aquí un ejemplo de distribución múltiple de un evento:

```
// Demostración de distribución múltiple de un evento.
using System;
// Declara un tipo delegado para un evento.
delegate void MiControladorEventos();
```

```
// Declara una clase que contiene un evento.
class MiEvento {
    public event MiControladorEventos UnEvento;

    // Esto es invocado para disparar el evento.
    public void Dispara() {
        if(UnEvento != null)
            UnEvento();
    }
}

class X {
    public void ControlaX() {
        Console.WriteLine("Evento recibido por objeto X");
    }
}

class Y {
    public void ControlaY() {
        Console.WriteLine("Evento recibido por objeto Y");
    }
}

class EventoDemo {
    static void Controlador() {
        Console.WriteLine("Evento recibido por EventoDemo");
    }

    static void Main() {
        MiEvento evt = new MiEvento();
        X xOb = new X();
        Y yOb = new Y();

        // Añade controladores a la lista de eventos.
        evt.UnEvento += Controlador;
        evt.UnEvento += xOb.ControlaX; ← Crea una cadena de distribución
        evt.UnEvento += yOb.ControlaY; múltiple para este evento.

        // Dispara el evento.
        evt.Dispara();
        Console.WriteLine();

        // Elimina el controlador.
        evt.UnEvento -= xOb.ControlaX;
        Console.WriteLine("Después de eliminar xOb.ControlaX");
        evt.Dispara();
    }
}
```

Los datos generados por este programa son:

```
Evento recibido por EventDemo
Evento recibido por objeto X
Evento recibido por objeto Y

Después de eliminar xOb.ControlaX
Evento recibido por EventDemo
Evento recibido por objeto Y
```

Este ejemplo crea dos clases adicionales, llamadas **X** y **Y**, que también definen controladores de eventos compatibles con **MiControladorEventos**. Así, estos controladores también se convierten en parte de la cadena de eventos. Observa que los controladores en **X** y **Y** no son estáticos. Eso significa que los objetos de cada uno deben ser creados, y el controlador vinculado a una instancia de objeto es añadido a la cadena de eventos. Cuando el evento se dispara, esos controladores en la cadena de eventos son invocados. Por lo mismo, si el contenido de la cadena de eventos se modifica, son invocados diferentes controladores. Esto se muestra en el programa cuando elimina **xOb.ControlaX**.

Debes comprender que los eventos se envían a objetos de instancia específicos, y no de manera genérica a una clase. Así, cada objeto de una clase debe registrarse para recibir una notificación del evento. Por ejemplo, el siguiente programa hace una distribución múltiple de un evento a tres objetos de tipo **X**:

```
// Los objetos, y no las clases, reciben los eventos.
using System;

// Declara un tipo delegado para un evento.
delegate void MiControladorEventos();

// Declara una clase que contiene un evento.
class MiEvento {
    public event MiControladorEventos UnEvento;

    // Esto es invocado para disparar el evento.
    public void Dispara() {
        if(UnEvento != null)
            UnEvento();
    }
}

class X {
    int id;

    public X(int x) { id = x; }

    public void ControlaX() {
        Console.WriteLine("Evento recibido por objeto " + id);
    }
}
```

```
class EventoDemo {
    static void Main() {
        MiEvento evt = new MiEvento();
        X o1 = new X(1);
        X o2 = new X(2);
        X o3 = new X(3);

        evt.UnEvento += o1.ControlaX;
        evt.UnEvento += o2.ControlaX; ← Cada objeto que quiere recibir un evento debe
        evt.UnEvento += o3.ControlaX; añadir su propio controlador a la cadena.

        // Dispara el evento.
        evt.Dispara();
    }
}
```

Los datos de salida generados por el programa son:

```
Evento recibido por objeto 1
Evento recibido por objeto 2
Evento recibido por objeto 3
```

Como lo muestran los datos de salida, cada objeto registra por separado su interés en un evento, y cada uno recibe la notificación por separado.

## Utilizar métodos anónimos con eventos

Los métodos anónimos son especialmente útiles cuando se trabaja con eventos porque pueden servir como controladores de estos últimos. Esto elimina la necesidad de declarar un método separado, lo que puede aclarar significativamente el código manejador de eventos. A continuación presentamos un ejemplo que utiliza un controlador de eventos anónimo:

```
// Utiliza un método anónimo como controlador de eventos.
using System;

// Declara un tipo delegado para un evento.
delegate void MiControladorEventos();

// Declara una clase que contiene un evento.
class MiEvento {
    public event MiControladorEventos UnEvento;

    // Esto se invoca para disparar el evento.
    public void Dispara() {
        if(UnEvento != null)
            UnEvento();
    }
}

class ControladorAnonMet {
    static void Main() {
        MiEvento evt = new MiEvento();
        // Utiliza un método anónimo como controlador de eventos.
```

```

    evt.UnEvento += delegate {
        // Éste es el controlador de eventos.
        Console.WriteLine("Evento recibido.");
    };

    // Dispara dos veces el evento.
    evt.Dispara();
    evt.Dispara();
}
}

```

Un método anónimo utilizado como controlador de eventos.

Los datos generados por el programa son:

```

Evento recibido.
Evento recibido.

```

En el programa, presta especial atención a la manera en que el controlador de eventos anónimo es añadido al evento por la siguiente secuencia de código:

```

// Utiliza un método anónimo como controlador de eventos.
evt.UnEvento += delegate {
    // Éste es el controlador de eventos.
    Console.WriteLine("Evento recibido.");
};

```

La sintaxis para utilizar un controlador de eventos anónimo es la misma que se usa para un método anónimo con cualquier otro tipo de delegado.

Los controladores de eventos anónimos son especialmente útiles porque, en muchas ocasiones, el controlador de eventos no es invocado por ningún otro código salvo el mecanismo que controla el evento. De esta manera, por lo general no existe una razón para utilizar un método independiente. Como se mencionó anteriormente en este mismo capítulo, C# 3.0 añade las expresiones lambda, que mejoran el concepto del método anónimo. En muchos casos, las expresiones lambda ofrecerán el mejor medio para implementar un controlador de eventos (ver capítulo 14).

## Pregunta al experto

**P:** ¿Existen técnicas especiales que deba utilizar para crear eventos o controladores de eventos que sean compatibles con los tipos de eventos utilizados en .NET Framework?

**R:** Sí. Aunque C# te permite escribir cualquier evento que deseas, por la compatibilidad de componentes con .NET Framework debes seguir los lineamientos de Microsoft a este respecto. En el centro de estos estatutos está el requerimiento de que los controladores de eventos tengan dos parámetros. El primero es una referencia al objeto que genera el evento. El segundo es un parámetro de tipo **EventArgs** que contiene cualquier otra información requerida por el controlador. Para eventos sencillos en los cuales el parámetro **EventArgs** no se utiliza, el tipo delegado puede ser **EventHandler**. Éste es un tipo predefinido que puede utilizarse para declarar eventos que no proporcionan información extra. Consulta las líneas guía de Microsoft al respecto para mayores detalles. Además, puedes encontrar información sobre la escritura de eventos compatibles con .NET Framework en mi libro *C# 3.0: Manual de referencia*.

## Nomenclaturas

Las nomenclaturas fueron mencionadas brevemente en el capítulo 1 porque es un concepto fundamental de C#. De hecho, cada programa escrito en C# hace uso de las nomenclaturas de una manera u otra. No teníamos necesidad de examinar con detalle las nomenclaturas durante los capítulos anteriores porque C# proporciona automáticamente una nomenclatura “global” predeterminada para cada programa.

Comencemos revisando lo que ya sabes sobre las nomenclaturas. Una *nomenclatura* define una región declarativa que proporciona un medio para mantener un conjunto de nombres separados de otros. Los nombres declarados en una nomenclatura no entrarán en conflicto con los mismos nombres declarados en otra. La nomenclatura utilizada por la biblioteca de .NET Framework (que es la misma de C#) es **System**. Por ello has incluido

```
using System;
```

al inicio de cada programa. Como viste en el capítulo 11, las clases E/S se definen dentro de una nomenclatura subordinada a **System** llamada **System.IO**. Existen muchas otras nomenclaturas subordinadas a **System** que contienen otras partes de la biblioteca de C#.

Las nomenclaturas son importantes porque ha habido una explosión de variables, métodos, propiedades y nombres de clases en los pasados años. Éstas incluyen rutinas de biblioteca, código de terceros y tu propio código. Sin las nomenclaturas, todos estos nombres competirían por su lugar en la nomenclatura global, lo que acarrearía infinidad de conflictos. Por ejemplo, si tu programa define una clase llamada **Finder**, podría entrar en conflicto con otra clase **Finder** proporcionada por una biblioteca de terceros que tu programa esté utilizando. Por fortuna, las nomenclaturas previenen este tipo de problemas porque una nomenclatura restringe la visibilidad de los nombres declarados dentro de ella.

## Declarar una nomenclatura

Una nomenclatura se declara utilizando la palabra clave **namespace**. El formato general de **namespace** se muestra a continuación:

```
namespace nombre {  
    // miembros  
}
```

Aquí, *nombre* es el nombre de la nomenclatura. Una declaración de nomenclatura define un alcance. Cualquier elemento declarado inmediatamente dentro de ella está en su radio de alcance a través de la misma. Dentro de una nomenclatura puedes declarar clases, estructuras, delegados, enumeraciones, interfaces y otras nomenclaturas.

He aquí un ejemplo de **namespace** que crea una nomenclatura llamada **Contador**. Localiza el nombre utilizado para implementar una sencilla clase de cuenta regresiva llamada **CuentaReg**.

```
// Declara una nomenclatura para contadores.  
namespace Contador { ← Declara una nomenclatura Contador.  
    // Una sencilla cuenta regresiva.  
    class CuentaReg {
```

```

int val;
public CuentaReg(int n) { val = n; }

public void Reset(int n) {
    val = n;
}

public int Count() {
    if(val > 0) return val--;
    else return 0;
}
}

} // Esto marca el final de la nomenclatura Contador.

```

Aquí, **CuentaReg** está declarada dentro del alcance definido por la nomenclatura **Contador**. Para seguir con el ejemplo, coloca el siguiente código dentro de un archivo llamado **Contador.cs**.

A continuación presentamos un programa que muestra el uso de la nomenclatura **Contador**:

```

// Muestra la nomenclatura Contador en acción.
using System;

class NSDemo {
    static void Main() {
        Contador.CuentaReg cd1 = new Contador.CuentaReg(10); ←
        int i;

        do {
            i = cd1.Count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();

        Contador.CuentaReg cd2 = new Contador.CuentaReg(20);

        do {
            i = cd2.Count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();
    }
}

```

Aquí, **Contador** califica  
**CuentaReg**.

Para compilar este programa debes incluir tanto el código anterior como el que contiene la nomenclatura **Contador**. Suponiendo que hayas llamado al código anterior **NSDemo.cs** y que hayas colocado el código fuente de la nomenclatura **Contador** en un archivo llamado **Contador.cs**, como se mencionó anteriormente, puedes utilizar la siguiente línea de comando para compilar el programa.

```
csc NSDemo.cs Contador.cs
```

Cuando ejecutas el programa produce el siguiente resultado:

```
10 9 8 7 6 5 4 3 2 1 0  
20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0  
4 3 2 1 0
```

Algunos aspectos importantes de este programa merecen especial atención. Primero, como **CuentaReg** está declarada dentro de la nomenclatura **Contador**, cuando se crea un objeto, **CuentaReg** debe ser calificada con **Contador**, como se muestra aquí:

```
Contador.CuentaReg cd1 = new Contador.CuentaReg(10);
```

Esta regla se puede generalizar. Cada vez que utilices un miembro de una nomenclatura, debes calificarlo con el nombre de la nomenclatura. Si no lo haces, el compilador no encontrará al miembro de esa nomenclatura.

En segundo lugar, una vez que un objeto de tipo **Cuenta** ha sido creado, no es necesario seguir calificándolo, ni a él ni a sus miembros, con el nombre de la nomenclatura. Así, **cd1.Count()** puede invocarse directamente sin calificación de nomenclatura, como lo demuestra la línea:

```
i = cd1.Count();
```

Tercero, aunque este ejemplo utiliza dos archivos separados, uno para almacenar la nomenclatura **Cuenta** y otro para el programa **NSDemo**, ambos pueden estar contenidos en el mismo archivo. Cuando termina la nomenclatura, se reanuda la nomenclatura exterior, que en este caso es la nomenclatura global. Más aún, un solo archivo puede contener dos o más nomenclaturas. Esta situación es similar a cuando un archivo contiene dos o más clases. Cada nomenclatura define su propia región declarativa.

## using

Como se explicó en el capítulo 1, si tu programa incluye referencias frecuentes a miembros de una nomenclatura, tener que especificarla cada vez que necesites hacer referencia rápida a ella resulta tedioso. La directiva **using** resuelve este problema. A lo largo de este libro has utilizado **using** para traer a la vista la nomenclatura **System**, por lo que ya estás familiarizado con ella. Como era de esperarse, **using** también puede utilizarse para traer a la vista nomenclaturas de tu propia creación.

Existen dos formatos de la directiva **using**. El primero es:

```
using nombre;
```

Aquí, *nombre* es el nombre de la nomenclatura que quieras accesar. Es el formato de **using** que ya has visto. Todos los miembros definidos dentro de la nomenclatura especificada son traídos a la vista y pueden utilizarse sin calificación. Se debe especificar una directiva **using** en la parte superior de cada archivo, antes que cualquier otra declaración o al inicio del cuerpo de la nomenclatura.

El siguiente programa es una versión modificada del ejemplo contador de la sección previa; muestra cómo puedes utilizar **using** para traer a la vista la nomenclatura de tu creación:

```
// Muestra una nomenclatura en acción.
using System;

// Trae Contador a la vista.
using Contador; ← using trae Contador a la vista.

class NSDemo {
    static void Main() {
        // Ahora, CuentaReg puede ser utilizada directamente.
        CuentaReg cd1 = new CuentaReg(10); ← Referencia directa a CuentaReg.
        int i;

        do {
            i = cd1.Count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();

        CuentaReg cd2 = new CuentaReg(20);

        do {
            i = cd2.Count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();
    }
}
```

Como antes, para compilar este programa debes incluir tanto el código anterior como el que contiene la nomenclatura **Contador**.

El programa ilustra un punto importante: el uso de una nomenclatura no anula otra. Cuando traes a la vista una nomenclatura, simplemente te permite utilizar sus nombres sin necesidad de calificación. Así, en el ejemplo, tanto **System** como **Contador** han sido traídos a la vista.

## Una segunda forma de using

La directiva **using** tiene un segundo formato que crea otro nombre, llamado alias, para una nomenclatura o un tipo. Este formato es:

```
using alias = nombre;
```

Aquí, *alias* se convierte en otro nombre para el tipo (como un tipo clase) o para la nomenclatura especificada por *nombre*. Una vez que se ha creado un alias, puede ser utilizado en lugar del nombre original.

En esta nueva versión del programa contador se utiliza un alias para **Contador.CuentaReg** llamado ahora **MiContador**:

```
// Muestra el uso de alias.
using System;

// Crea un alias para Contador.CuentaReg.
using MiContador = Contador.CuentaReg; ← Crea un alias para
  Contador.CuentaReg.

class NSDemo {
    static void Main() {
        MiContador cd1 = new MiContador(10); ← Usa el alias.
        int i;

        do {
            i = cd1.Count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();

        MiContador cd2 = new MiContador(20);

        do {
            i = cd2.Count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();

        cd2.Reset(4);
        do {
            i = cd2.Count();
            Console.Write(i + " ");
        } while(i > 0);
        Console.WriteLine();
    }
}
```

En este caso, **MiContador** es un alias para **Contador.CuentaReg**. Una vez que **MiContador** ha sido especificado como un alias, puede utilizarse para declarar objetos sin posteriores calificaciones de nomenclatura. Por ejemplo, en el programa, la línea

```
MiContador cd1 = new MiContador(10);
```

crea un objeto **CuentaReg**.

## Las nomenclaturas son aditivas

Puede haber más de una declaración de nomenclatura del mismo nombre. Esto permite que una nomenclatura sea dividida en varios archivos o incluso en trozos separados dentro del mismo archivo. Por ejemplo, el siguiente código especifica la nomenclatura **Contador**. Añade una clase llamada **CuentaPro**, que cuenta hacia adelante en lugar de hacerlo hacia atrás.

```
// He aquí otra parte de la nomenclatura Contador.
namespace Contador { ← Añadidura a la nomenclatura Contador.
    // Un contador progresivo simple.
    class CuentaPro {
        int val;
        int target;

        public int Target { get{ return target; } }

        public CuentaPro(int n) { target = n; val = 0; }

        public void Reset(int n) {
            target = n;
            val = 0;
        }

        public int Count() {
            if(val < target) return val++;
            else return target;
        }
    }
}
```

Para continuar con el ejemplo, coloca el código inmediato anterior en un archivo llamado **Contador2.cs**.

El siguiente programa muestra el efecto aditivo de las nomenclaturas al utilizar **CuentaReg** y **CuentaPro**:

```
// Las nomenclaturas son aditivas.
using System;

// Trae a la vista la nomenclatura Contador completa.
using Contador;
class NSDemo {
```

```
static void Main() {
    CuentaReg cd = new CuentaReg(10);
    CuentaPro cu = new CuentaPro(8);
    int i;

    do {
        i = cd.Count();
        Console.Write(i + " ");
    } while(i > 0);
    Console.WriteLine();

    do {
        i = cu.Count();
        Console.Write(i + " ");
    } while(i < cu.Target);

}
```

Para compilar este programa, debes incluir el código inmediato anterior con los archivos que contienen la nomenclatura **Contador**. Suponiendo que hayas llamado al código anterior **NSDemo.cs** y que los archivos que contienen la nomenclatura **Contador** se llamen **Contador.cs** y **Contador2.cs**, puedes utilizar la siguiente línea de comando para compilar el programa:

```
csc NSDemo.cs Contador.cs Contador2.cs
```

El programa arroja los siguientes datos de salida:

```
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8
```

Observa una cosa: la directiva

```
using Contador;
```

trae a la vista el contenido completo de la nomenclatura **Contador**. De esta manera, se puede hacer referencia directa tanto a **CuentaReg** como a **CuentaPro**, sin que se califiquen con el nombre de la nomenclatura. No importa que la nomenclatura **Contador** esté dividida en dos archivos.

## Las nomenclaturas pueden anidarse

Una nomenclatura puede anidarse dentro de otra. Cuando se hace referencia a una nomenclatura anidada desde código fuera de la anidación, tanto la nomenclatura externa como la interna deben ser especificadas, separando ambas con un punto. Para comprender el proceso, analiza el siguiente programa:

```
// Las nomenclaturas pueden anidarse.
using System;
namespace Nomenclatural1 {
    class ClaseA {
        public ClaseA() {
```

```

        Console.WriteLine("construyendo ClaseA");
    }
}

namespace Nomenclatura2{ // una nomenclatura anidada ←
    class ClaseB {
        public ClaseB() {
            Console.WriteLine("construyendo ClaseB");
        }
    }
}

class AnidaDemo {
    static void Main() {
        Nomenclatural.ClaseA a= new Nomenclatural.ClaseA();

        // Nomenclatura2.ClaseB b = new Nomenclatura2.ClaseB(); // ;Error!
        Nomenclatura2 no es visible

        Nomenclatural.Nomenclatura2.ClaseB b = new Nomenclatural.
        Nomenclatura2.ClaseB(); // esto es correcto
    }
}
}

```

Los datos generados por este programa son los siguientes:

```
construyendo ClaseA
construyendo ClaseB
```

En este programa, la nomenclatura **Nomenclatura2** está anidada dentro de **Nomenclatura1**. De esta manera, para hacer referencia a **ClaseB** desde código que se encuentra fuera de ambas nomenclaturas, debes calificar la referencia con los nombres de ambas nomenclaturas. El nombre de la **Nomenclatura2** es insuficiente. Como se explicó anteriormente, se deben utilizar los nombres de ambas nomenclaturas separados por un punto. Por lo mismo, para hacer referencia a la **ClaseB** dentro de **Main( )**, debes utilizar el formato **Nomenclatura1.Nomenclatura2.ClaseB**.

Las nomenclaturas pueden anidarse en más de dos niveles. Cuando esto sucede, un miembro que se encuentre dentro de la anidación debe ser calificado con todos los nombres de las nomenclaturas que lo envuelven.

Puedes especificar una nomenclatura anidada utilizando una sola declaración **namespace**, separando cada nomenclatura con un punto. Por ejemplo:

```
namespace NomenclaturaExterna {
    namespace NomenclaturaInterna {
        // ...
    }
}
```

también puede ser especificada de la siguiente manera:

```
namespace NomenclaturaExterna.NomenclaturaInterna {
    // ...
}
```

## La nomenclatura global

Si no declaras una nomenclatura para tu programa, se utiliza la nomenclatura global predeterminada. Por eso no tuviste necesidad de utilizar **namespace** en los programas de los capítulos anteriores. Aunque la nomenclatura global es conveniente para los programas cortos y de ejemplo que se encuentran en este libro, la mayoría del código en el mundo real estará contenido dentro de una nomenclatura propiamente declarada. La principal razón para encapsular tu código dentro de una nomenclatura declarada es que previene los conflictos de nombre. Las nomenclaturas son otra herramienta con la que cuentas para ayudarte a organizar programas y hacerlos viables en el actual y complejo ambiente de redes.

### Pregunta al experto

**P:** Leyendo la documentación que acompaña a la Edición Express de Visual C# encontré una característica llamada **calificador alias de nomenclaturas**, ¿qué significa?

**R:** Aunque las nomenclaturas ayudan a prevenir conflictos de nombres, no los eliminan por completo. Una manera en que un conflicto aún puede ocurrir es cuando se declara el mismo nombre dentro de dos diferentes nomenclaturas y luego se intenta traer ambos a la vista. Por ejemplo, supongamos que tienes dos nomenclaturas, una llamada **Alfa** y la otra **Beta**. Supongamos también que ambas tengan una clase llamada **MiClase**. Si quieras traer ambas a la vista utilizando la declaración **using**, **MiClase** de la nomenclatura **Alfa** entrará en conflicto con **MiClase** de la nomenclatura **Beta**, provocando un error de ambigüedad. En esta situación puedes utilizar el *calificador alias de nomenclaturas* :: para especificar explícitamente a qué nomenclatura te refieres.

El operador :: tiene el siguiente formato general:

*nomenclatura-alias::identificador*

Aquí, *nomenclatura-alias* es el alias de la nomenclatura e *identificador* es el nombre del miembro de esa nomenclatura. Por ejemplo, retomando el caso de las nomenclaturas **Alfa** y **Beta** que acabamos de describir, la siguiente declaración **using** crea un alias llamado **alfa** para la nomenclatura **Alfa**:

```
using alfa = Alfa;
```

Después de que se ha compilado esta declaración, la referencia que aparece a continuación alude a la versión de **MiClase** dentro de la nomenclatura **Alfa**, no dentro de la nomenclatura **Beta**:

```
alpha::MiClase
```

De esta manera, el calificador alias de nomenclaturas resuelve una situación ambigua.

Un último punto: puedes utilizar el calificador :: para hacer referencia a nomenclaturas globales utilizando el identificador predefinido **global**, como en

```
global::Test
```

**Prueba esto****Colocar Set en una nomenclatura**

En la sección *Prueba esto: crear una clase conjunto* del capítulo 7, creaste una clase que implementa un tipo conjunto llamada **Conjunto**. Éste es precisamente el tipo de clase que debes considerar para colocar en tu propia nomenclatura. Una razón para esto es que el nombre **Conjunto** puede entrar fácilmente en conflicto con otras clases del mismo nombre. Pero colocando **Conjunto** en su propia nomenclatura, puedes evitar conflictos de nombre potenciales. Este ejemplo examina el proceso.

**Paso a paso**

1. Utilizando el código de **ConjuntoDemo.cs** mostrado en el capítulo 7, mueve la clase entera **Conjunto** a un archivo llamado **Conjunto.cs**. En el proceso, encierra el código **Conjunto** dentro de la nomenclatura **MisTipos**, como se muestra aquí:

```
// Coloca la clase Conjunto en su propia nomenclatura.
using System;

namespace MisTipos {
    class Conjunto {
        char[] miembros; // éste es el arreglo que contiene el conjunto

        // Una propiedad Longitud autoimplementada, sólo-lectura.
        public int Longitud { get; private set; }

        // Construye un conjunto nulo.
        public Conjunto() {
            Longitud = 0;
        }

        // Construye un conjunto vacío dado su tamaño.
        public Conjunto(int tamaño) {
            miembros = new char[tamaño]; // reserva memoria para el conjunto
            Longitud = 0; // ningún miembro cuando se construye
        }

        // Construye un conjunto a partir de otro.
        public Conjunto(Conjunto s) {
            miembros = new char[s.Longitud]; // reserva memoria para el
            conjunto
            for(int i=0; i < s.Longitud; i++) miembros[i] = s[i];
            Longitud = s.Longitud; // número de miembros
        }

        // Implementa un indexador sólo-lectura.
        public char this[int idx] {
            get {

```

(continúa)

```
        if(idx >= 0 & idx < Longitud) return miembros[idx];
        else return (char)0;
    }
}

/* Verifica si un elemento está en el conjunto.
   Regresa el índice del elemento o -1 si no se localiza. */
int find(char ch) {
    int i;

    for(i=0; i < Longitud; i++)
        if(miembros[i] == ch) return i;

    return -1;
}

// Añade un elemento único al conjunto.
public static Conjunto operator +(Conjunto ob, char ch) {

    // Si ch ya está en el conjunto, regresa una copia del
    // conjunto original.
    if(ob.find(ch) != -1) {

        // Regresa una copia del conjunto original.
        return new Conjunto(ob);

    } else { // Regresa un conjunto nuevo que contiene un elemento
        nuevo.

        // Hace que el conjunto sea un elemento más largo que el
        // original.
        Conjunto newset = new Conjunto(ob.Longitud + 1);

        // Copia elementos a un nuevo conjunto.
        for(int i=0; i < ob.Longitud; i++)
            newset.miembros[i] = ob.miembros[i];

        // Establece la propiedad Longitud.
        newset.Longitud = ob.Longitud+1;

        // Añade un nuevo elemento a un conjunto nuevo.
        newset.miembros[newset.Longitud-1] = ch;

        return newset; // regresa el nuevo conjunto.
    }
}
```

```
// Elimina un elemento de un conjunto.
public static Conjunto operator -(Conjunto ob, char ch) {
    Conjunto newset = new Conjunto();
    int i = ob.find(ch); // i será -1 si no se localiza el elemento.

    // Copia y compara los elementos restantes.
    for(int j=0; j < ob.Longitud; j++)
        if(j != i) newset = newset + ob.miembros[j];

    return newset;
}

// Unión de conjuntos.
public static Conjunto operator +(Conjunto ob1, Conjunto ob2) {
    Conjunto newset = new Conjunto(ob1); // copia el primer conjunto

    // Añade elementos únicos al segundo conjunto.
    for(int i=0; i < ob2.Longitud; i++)
        newset = newset + ob2[i];

    return newset; // regresa el conjunto actualizado
}

// Establece diferencia.
public static Conjunto operator -(Conjunto ob1, Conjunto ob2) {
    Conjunto newset = new Conjunto(ob1); // copia el primer conjunto

    // Resta elementos del segundo conjunto.
    for(int i=0; i < ob2.Longitud; i++)
        newset = newset - ob2[i];

    return newset; // regresa el conjunto actualizado
}
}
} // El final de la nomenclatura MisTipos.
```

- 2.** Después de poner **Conjunto** en la nomenclatura **MisTipos**, necesitas incluir la nomenclatura **MisTipos** en cualquier programa que utilice **Conjunto**, como se muestra aquí:

```
using MisTipos;
```

- 3.** De manera alternativa, puedes calificar completamente las referencias a **Conjunto**, como se muestra en el siguiente ejemplo:

```
MisTipos.Conjunto s1 = new MisTipos.Conjunto();
```

## ✓ Autoexamen Capítulo 12

- 1.** Muestra cómo se declara un delegado llamado **Filtro** que regresa un **double** y toma un argumento **int**.
- 2.** ¿Cómo se realiza una transmisión múltiple usando un delegado?
- 3.** ¿Qué es un método anónimo?
- 4.** ¿Un método anónimo puede tener parámetros? ¿Puede regresar un valor?
- 5.** ¿Cómo se relacionan los delegados y los eventos?
- 6.** ¿Qué palabra clave declara un evento?
- 7.** ¿Un evento es sujeto a una transmisión múltiple?
- 8.** ¿Un evento se envía a una instancia o a una clase?
- 9.** ¿Cuál es el principal beneficio de las nomenclaturas?
- 10.** Muestra el formato alias de **using**.
- 11.** Muestra otra forma de declarar esta nomenclatura:

```
namespace X {  
    namespace Y {  
        // ...  
    }  
}
```

# Capítulo 13

## Genéricos

## Habilidades y conceptos clave

- Fundamentos de los genéricos
- Las limitaciones de la clase base
- Las limitaciones de la interfaz
- Las limitaciones del constructor
- Las limitaciones del tipo referencia
- Las limitaciones del tipo valor
- El operador **default**
- Estructuras genéricas
- Métodos genéricos
- Delegados genéricos
- Interfaces genéricas

---

**U**n evento extremadamente importante ocurrió en el ciclo de vida de C# cuando se liberó la versión 2.0. Fue la adición de *genéricos*. Los genéricos no sólo añadieron un nuevo elemento sintáctico a C#, también trajeron como consecuencia muchos cambios y actualizaciones a la biblioteca.NET. Aunque ya han pasado varios años desde la adición de los genéricos, sus efectos aún resuenan a través de C# 3.0. Los genéricos se han convertido en una parte indispensable de la programación en C#.

La característica *genéricos* hace posible crear clases, interfaces, métodos y delegados que funcionan en modo tipo-seguro con diferentes clases de datos. Como probablemente sepas, muchos algoritmos son iguales desde la perspectiva lógica, sin importar el tipo de datos que se les aplique. Por ejemplo, el mecanismo que soporta una organización en cola es el mismo sin importar que la cola esté almacenando tipos de datos **int**, **string** u **object**, o clases definidas por el usuario. Antes de los genéricos, tal vez tendrías que crear varias versiones diferentes del mismo algoritmo para manejar diversos tipos de datos. A través del uso de genéricos, puedes definir una sola vez cierta solución, independientemente del tipo específico de datos, y luego aplicar esa solución a una gran variedad de tipos de datos sin esfuerzo adicional.

Antes de comenzar, es necesario dejar en claro que la característica genéricos es una de las más sofisticadas y, en ocasiones, más complicadas de C#. Más aún, el tema de los genéricos es muy largo. No es posible detallar todos los aspectos de este poderoso subsistema en esta guía para principiantes. En lugar de ello, este capítulo presenta una introducción a la teoría de los genéricos, describe su sintaxis y luego muestra varios ejemplos que los ponen en acción. Al concluir este capítulo, estarás en condiciones de comenzar a escribir tu propio código genérico y hacer uso de las características genéricas de la biblioteca.NET. También contarás con los conocimientos necesarios para moverte entre sus características más avanzadas.

## ¿Qué son los genéricos?

En esencia, el término *genéricos* significa *tipos parametrizados*. Los tipos parametrizados son importantes porque te permiten crear clases, estructuras, interfaces, métodos y delegados en los cuales el tipo de datos sobre los que operan se especifica como parámetro. Al utilizar genéricos es posible crear una sola clase, por ejemplo, que automáticamente funcione con diferentes tipos de datos. Una clase, estructura, interfaz, método o delegado que opera sobre tipos parametrizados recibe el nombre de *genérico*, como *clase genérica* o *método genérico*.

Es importante comprender que C# siempre ha ofrecido la posibilidad de crear clases, estructuras, interfaces, métodos y delegados generalizados operando a través de referencias tipo **object**. Como **object** es la clase base para todos los tipos de datos, una referencia **object** puede dirigirse hacia cualquier tipo de objeto. Así, en el código pregenérico, el código generalizado utilizaba referencias **object** para operar sobre una variedad de diferentes objetos. El problema es que no podía hacerlo con seguridad de tipo porque se necesitaba aplicar transformación para convertir el tipo **object** al tipo real de dato requerido por la aplicación. Los genéricos añaden la seguridad de tipo de que carecía su antecesor porque ya no es necesario emplear la transformación para traducir entre **object** y el tipo de dato real. Esto clarifica tu código. También expande la oportunidad de reutilizarlo.

## Fundamentos de los genéricos

Aunque la característica genéricos puede aplicarse a varios constructores C#, su uso principal es crear clases genéricas. Por ello, será ahí donde iniciemos su estudio. Comencemos con un ejemplo sencillo. El siguiente programa define dos clases. La primera es una clase genérica llamada **MiClaseGen** y la segunda es **GenericosDemo**, que utiliza la primera.

```
// Una sencilla clase genérica.
using System;

// Aquí, MiClaseGen es una clase genérica que tiene un tipo
// parámetro llamado T. T será reemplazado por el tipo real
// cuando se construye un objeto de MiClaseGen.
class MiClaseGen<T> { ← Declara una clase genérica.
    T ob; // declara una variable de tipo T ←

    // Observa que este constructor tiene un parámetro de tipo T.
    public MiClaseGen(T o) { ←
        ob = o;
    }

    // Regresa ob, que es de tipo T.
    public T GetOb() { ←
        return ob;
    }
}
```

Observa el uso del parámetro de tipo **T**.

```

// Muestra la clase genérica.
class GenericosDemo {
    static void Main() {
        // Declara una referencia MiClaseGen para int.
        MiClaseGen<int> iOb; ← Crea una referencia MiClaseGen,  

                                transmitiendo int a T.
        // Crea un objeto MiClaseGen<int> y asigna a iOb una referencia a él.
        iOb = new MiClaseGen<int>(88); ← Crea un objeto MiClaseGen,  

   otra vez transmite int a T.
        // Obtiene el valor en iOb.
        int v = iOb.GetOb();
        Console.WriteLine("iOb es una instancia de MiClaseGen<int>.\n" +
                           "El valor regresado por GetOb(): " + v + "\n");
    }

    // Crea un objeto MiClaseGen para cadenas de caracteres.
    MiClaseGen<string> strOb = new MiClaseGen<string>("Genéricos Demo");
    ↑
    // Obtiene el valor en strOb.
    string str = strOb.GetOb(); ← Crea una referencia MiClaseGen  

                                y un objeto que utiliza string.
    Console.WriteLine("strOb es una instancia de MiClaseGen<string>.\n" +
                           "El valor regresado por GetOb(): " + str + "\n");
}
}

```

Los datos generados por el programa son:

```
iOb es una instancia de MiClaseGen<int>.  
El valor regresado por GetOb(): 88
```

```
strOb es una instancia de MiClaseGen<string>.  
El valor regresado por GetOb(): Genéricos Demo
```

Examinemos con cuidado este programa. Primero, observa cómo se declara **MiClaseGen** por la siguiente línea:

```
class MiClaseGen<T> {
```

Aquí, **T** es el nombre de un *parámetro de tipo*. Este nombre es utilizado como un marcador de posición para el verdadero tipo que será especificado cuando se cree un objeto **MiClaseGen**. Así, **T** es utilizada dentro de **MiClaseGen** cada vez que se requiere el parámetro de tipo. Observa que **T** está encerrada entre <>. Esta sintaxis puede ser generalizada. Cada vez que se declara un parámetro de tipo, se especifica entre corchetes angulados. Como **MiClaseGen** utiliza un parámetro de tipo, se le considera una *clase genérica*.

En la declaración de **MiClaseGen**, no hay un significado especial para el nombre **T**. Se pudo utilizar cualquier identificador válido, pero **T** es tradicional. Otros parámetros tipo comúnmente utilizados son **V** y **E**. Por supuesto, también puedes utilizar nombres descriptivos para parámetros tipo, como **TValor** o **TClave**. Cuando se utiliza un nombre descriptivo, es una práctica común utilizar **T** como la primera letra.

A continuación, **T** se utiliza para declarar una variable llamada **ob**, como se muestra aquí:

```
T ob; // declara una variable de tipo T
```

Como se explicó, **T** es un marcador de posición para el tipo verdadero que será especificado cuando se cree un objeto **MiClaseGen**. Así, **ob** será una variable de tipo *atado a T* cuando se inicialice el objeto **MiClaseGen**. Por ejemplo, si se especifica un tipo **string** para **T**, en esa instancia, **ob** será de tipo **string**.

Ahora analiza el constructor de **MiClaseGen**.

```
public MiClaseGen(T o) {
    ob = o;
}
```

Observa que su parámetro, **o**, es de tipo **T**. Esto significa que el tipo verdadero de **o** es determinado por el tipo al cual **T** está atada cuando se crea el objeto **MiClaseGen**. Además, como tanto el parámetro **o** como la variable de instancia **ob** son de tipo **T**, ambas serán del tipo real cuando se cree el objeto **MiClaseGen**.

El parámetro de tipo **T** también puede utilizarse para especificar el tipo de regreso de un método, como en el caso del método **GetOb()** que se muestra a continuación:

```
public T GetOb() {
    return ob;
}
```

Como **ob** también es del tipo **T**, su tipo es compatible con el tipo de regreso especificado en **GetOb()**.

La clase **GenericosDemo** muestra el funcionamiento de la clase **MiClaseGen**. Primero crea una versión de **MiClaseGen** para tipo **int**, como se muestra aquí:

```
MiClaseGen<int> iOb;
```

Observa con cuidado esta declaración. Primero, advierte que el tipo **int** se especifica dentro de corchetes angulados después de **MiClaseGen**. En este caso, **int** es un *argumento de tipo* que está atado a **T**, el parámetro de tipo de **MiClaseGen**. Esto crea una versión de **MiClaseGen** en la cual todos los usos de **T** serán reemplazados por un **int**. Así, para esta declaración, **ob** es de tipo **int**, y el tipo de regreso de **GetOb()** es de tipo **int** también.

Cuando especificas un argumento de tipo como **int** o **string** para **MiClaseGen**, estás creando lo que en C# se llama un *tipo de construcción cerrada*. Así, **MiClaseGen<int>** es un tipo de construcción cerrada. En esencia, un tipo genérico, como **MiClaseGen<T>**, es una abstracción; sólo después de construir una versión específica, como **MiClaseGen<int>**, se crea un tipo concreto. En la terminología de C#, un constructor como **MiClaseGen<T>** es llamado *tipo de construcción abierta*, porque se especifica **T** (en lugar de un tipo real como **int**).

La siguiente línea asigna a **iOb** la referencia a una instancia correspondiente a la versión **int** de la clase **MiClaseGen**.

```
iOb = new MiClaseGen<int>(88);
```

Observa que cuando se invoca al constructor **MiClaseGen**, también el argumento de tipo **int** es especificado. Esto es necesario porque el tipo de variable (en este caso **iOb**) a la cual le es asignada

la referencia es **MiClaseGen<int>**. De esta manera, la referencia regresada por **new** también tiene que ser del tipo **MiClaseGen<int>**. De lo contrario se generará un error en tiempo de compilación. Por ejemplo, la siguiente asignación causará un error de compilación.

```
iOb = new MiClaseGen<byte>(16); // ¡Error! ¡Tipo erróneo!
```

Como **iOb** es de tipo **MiClaseGen<int>**, no puede ser utilizada para hacer referencia a un objeto de **MiClaseGen<byte>**. Esta verificación de tipos es una de las principales ventajas de los genéricos porque garantiza la seguridad de los tipos.

A continuación, el programa obtiene el valor de **ob** utilizando la siguiente línea:

```
int v = iOb.GetOb();
```

Como el tipo de regreso de **GetOb()** es **T**, que fue reemplazado por **int** cuando **iOb** fue declarada, el tipo de regreso a esta invocación de **GetOb()** es también **int**. Así, este valor puede ser asignado a una variable **int**.

A continuación, **GenericosDemo** declara un objeto de tipo **MiClaseGen<string>**.

```
MiClaseGen<string> strOb = new MiClaseGen<string>("Genéricos Demo.");
```

Como el argumento de tipo es **string**, dentro de **MiClaseGen** **string** es sustituido por **T**. Esto crea una versión **string** de **MiClaseGen**, como lo demuestran las líneas restantes del programa.

## Los tipos genéricos difieren con base en sus argumentos de tipo

Un punto clave para comprender los tipos genéricos es que la referencia a una versión específica de un tipo genérico no es de tipo compatible con otra versión del mismo tipo genérico. Por ejemplo, retomando el programa que acabamos de analizar, la siguiente línea de código es errónea y no se compilará.

```
iOb = strOb; // ¡Error!
```

Aunque tanto **iOb** como **strOb** son del tipo **MiClaseGen<T>**, cada una hace referencia a un tipo diferente porque sus argumentos de tipo difieren entre sí.

## Los genéricos mejoran la seguridad de los tipos

En este punto es probable que te estés haciendo esta pregunta: dado que la misma funcionalidad encontrada en la clase **MiClaseGen** puede obtenerse sin los genéricos, simplemente especificando **object** como el tipo de dato y empleando las transformaciones requeridas, ¿cuál es el beneficio de hacer **MiClaseGen** un genérico? La respuesta es que los genéricos aseguran automáticamente la seguridad de tipos en todas las operaciones que invocan **MiClaseGen**. En el proceso, los genéricos eliminan la necesidad de que utilices transformaciones y que escribas el código necesario para verificar los tipos.

Para comprender los beneficios que ofrecen los genéricos, primero analiza el siguiente programa que crea un equivalente no genérico de **MiClaseGen** llamado **NoGenerico**:

```

// NoGenerico es un equivalente funcional a MiClaseGen
// pero no utiliza genéricos.
using System;
class NoGenerico {
    object ob; // ob es ahora de tipo object ← Utiliza una referencia object.
    // Transmite al constructor una referencia de tipo object.
    public NoGenerico(object o) {
        ob = o;
    }
    // Regresa el tipo object.
    public object GetOb() {
        return ob;
    }
}

// Muestra el funcionamiento de la clase sin genéricos.
class NoGenDemo {
    static void Main() {
        NoGenerico iOb;

        // Crea un objeto NoGenerico.
        iOb = new NoGenerico(88);

        // Obtiene el valor en iOb.
        // Esta vez es necesario realizar una transformación.
        int v = (int) iOb.GetOb(); ← Transformación necesaria.
        Console.WriteLine("iOb es una instancia de NoGenerico.\n" +
            "Por tanto, el valor regresado por GetOb() " +
            "es object.\nDebe ser transformado en int: " +
            v + "\n");

        // Crea otro objeto NoGenerico y almacena un string en él.
        NoGenerico strOb = new NoGenerico("Clase NoGenerico");

        // Obtiene el valor de strOb.
        // Otra vez, advierte que es necesaria la trasformación.
        String str = (string) strOb.GetOb();← Transformación necesaria.
        Console.WriteLine("strOb es una instancia de NoGenerico.\n" +
            "Por tanto, el valor regresado por GetOb() " +
            "es también object.\n Debe ser transformado en
            string: " + str + "\n");

        // Esto compila, pero conceptualmente ¡es un error!
        iOb = strOb;

        // La siguiente línea genera una excepción en tiempo de ejecución.
        // v = (int) iOb.GetOb(); // ¡error en tiempo de ejecución!
    }
}

```

↑  
Desajuste en tiempo de ejecución.

Este programa produce lo siguiente:

```
iOb es una instancia de NoGenerico.  
Por tanto, el valor regresado por GetOb() es object.  
Debe ser transformado en int: 88  
  
strOb es una instancia de NoGenerico.  
Por tanto, el valor regresado por GetOb() es también object.  
Debe ser transformado en string: Clase NoGenerico.
```

Hay muchas cosas interesantes en esta versión. Primero, observa que **NoGenerico** reemplaza todos los usos de **T** con **object**. Esto permite que **NoGenerico** sea capaz de almacenar cualquier tipo de objeto, lo mismo que la versión genérica. Sin embargo, esto tiene muchos inconvenientes por dos razones; la primera de ellas es que se tiene que emplear transformación explícita para recuperar los datos almacenados; la segunda es que muchos tipos de datos sufren errores de desajuste y no pueden localizarse hasta la ejecución del programa. Veamos de cerca cada uno de estos problemas.

Primero, observa esta línea:

```
int v = (int) iOb.GetOb();
```

Como el tipo de regreso de **GetOb()** es ahora **object**, es necesario hacer una transformación a **int** para permitir que el valor regresado por **GetOb()** sea desencajonado y almacenado en **v**. Si eliminas la transformación, el programa no compilará. En la versión genérica del programa, esta transformación no fue necesaria porque **int** fue especificado como el argumento de tipo cuando se construyó **iOb**. En la versión no genérica, se debe emplear la transformación. Esto no sólo es un inconveniente, sino una fuente potencial de errores.

Ahora considera la siguiente secuencia que se localiza cerca del final del programa:

```
// Esto compila, pero conceptualmente ¡es un error!  
iOb = strOb;  
  
// La siguiente línea genera una excepción en tiempo de ejecución.  
// v = (int) iOb.GetOb(); // ¡error en tiempo de ejecución!
```

Aquí, **strOb** es asignada a **iOb**. Sin embargo, **strOb** hace referencia a un objeto que contiene una cadena de caracteres, no un entero. Esta asignación es sintácticamente válida porque todas las referencias **NoGenerico** son del mismo tipo. De esta manera, cualquier referencia **NoGenerico** puede hacer referencia a cualquier objeto **NoGenerico**. No obstante, la declaración es semánticamente errónea, como lo muestran las líneas comentadas. En esa línea, el tipo de regreso de **GetOb()** es transformado en **int** y luego se hace un intento por asignar este valor a **v**. El problema es que **iOb** ahora hace referencia a un objeto que almacena un **string**, no un **int**. Desafortunadamente, sin el uso de genéricos, el compilador no detectará este error. En lugar de ello, ocurrirá una excepción en tiempo de ejecución cuando se intente realizar la transformación a **int**. Para que lo compruebes tú mismo, elimina los símbolos de comentario al principio de la línea, compila y ejecuta el programa. Ocurrirá un error en tiempo de ejecución.

La secuencia anterior no puede ocurrir cuando se utilizan los genéricos. Si esta misma secuencia se ejecuta en la versión genérica del programa, el compilador la detectará y reportará el error, previniendo así una equivocación seria que devendría en una excepción en tiempo de ejecución. La

capacidad para crear código de tipo seguro en el cual los errores de desajustes de tipo son capturados durante el tiempo de compilación es una ventaja clave de los genéricos. Aunque el uso de referencias **object** para crear un código “genérico” siempre ha sido posible en C#, el código no tenía seguridad de tipos y su uso erróneo podría resultar en excepciones durante la ejecución del programa. Los genéricos previenen que ocurran este tipo de errores. En esencia, a través de los genéricos, lo que fueron errores en tiempo de ejecución se convierten en errores en tiempo de compilación. Esto es una ventaja muy grande.

## Una clase genérica con dos parámetros de tipo

Puedes declarar más de un parámetro de tipo en un tipo genérico. Para especificar dos o más parámetros de tipo, simplemente utiliza una lista separada por comas. Por ejemplo, la siguiente clase **DosGen** es una variación de **MiClaseGen** que tiene dos parámetros de tipo:

```
// Una sencilla clase genérica con dos parámetros de tipo: T y V.
using System;

// DosGen tiene dos parámetros de tipo, T y V.
class DosGen<T, V> { ← Utiliza dos parámetros de tipo.
    T ob1;
    V ob2;

    // Observa que este constructor utiliza los dos parámetros de tipo, T y V.
    public DosGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    public T GetOb1() {
        return ob1;
    }

    public V GetOb2() {
        return ob2;
    }
}

// Muestra dos parámetros de tipo genéricos.
class SimpGen {
    static void Main() {

        Console.WriteLine("Construye un objeto DosGen<int, string>.");
        DosGen<int, string> tgObj =
            new DosGen<int, string>(1024, "Utiliza dos parámetros de tipo");

        // Obtiene y muestra los valores.
        int v = tgObj.GetOb1();
        Console.WriteLine("El valor de ob1: " + v); ↑ Transmite dos argumentos de tipo.

        string str = tgObj.GetOb2();
        Console.WriteLine("El valor de ob2: " + str);
    }
}
```

Los datos generados por el programa son:

```
Construye un objeto DosGen<int, string>.
El valor de ob1: 1024
El valor de ob2: Utiliza dos parámetros de tipo
```

Observa cómo se declara **DosGen**:

```
class DosGen<T, V> {
```

Especifica dos parámetros de tipo, **T** y **V**, separados por una coma. Como tiene dos parámetros de tipo, **DosGen** debe especificar dos argumentos de tipo cuando se crea un objeto, como se muestra a continuación:

```
DosGen<int, string> tgObj =
    new DosGen<int, string>(1024, "Utiliza dos parámetros de tipo");
```

En este caso, **int** es sustituido por **T** y **string** es sustituida por **V**.

Aunque los dos argumentos de tipo son diferentes en este ejemplo, es posible que ambos tipos sean el mismo. Por ejemplo, la siguiente línea de código es válida:

```
DosGen<double, double> x = new DosGen<double, double>(98.6, 102.4);
```

En este caso, tanto **T** como **V** son de tipo **double**. Por supuesto, si los argumentos de tipo fueran siempre el mismo, serían innecesarios los dos parámetros de tipo.

La sintaxis de los genéricos mostrada en los ejemplos anteriores se puede generalizar. He aquí la sintaxis para declarar una clase genérica:

```
class nombre-clase<lista-parámetros-de-tipo> { // ...
```

He aquí la sintaxis para declarar una referencia hacia una clase genérica y darle un valor inicial:

```
nombre-clase<lista-argumentos-de-tipo> nombre-var =
    new nombre-clase<lista-argumentos-de-tipo>(lista-argumentos-cons);
```

## Pregunta al experto

**P:** Usted afirma que clases, estructuras, métodos, interfaces y delegados pueden ser genéricos. ¿Qué hay de las propiedades, operadores, indexadores y eventos?

**R:** Propiedades, operadores, indexadores y eventos no pueden declarar parámetros de tipo. Por ello, no pueden ser genéricos. No obstante, pueden utilizarse en una clase genérica y hacer uso de los parámetros de tipo definidos por esa clase.

## Tipos limitados

En los ejemplos anteriores, los parámetros de tipo podían ser reemplazados por cualquier tipo. Por ejemplo, dada la siguiente declaración:

```
class MiClaseGen<T> {
```

T puede especificar cualquier tipo. Así, es legal crear objetos **MiClaseGen** en los cuales T es reemplazada por **int**, **double**, **string**, **FileStream** o cualquier otro tipo. Aunque no tener restricciones en los argumentos de tipo es benéfico para muchos propósitos, algunas veces es útil limitar los tipos que pueden ser atados a un parámetro de tipo. Por ejemplo, es posible que quieras crear un método que opere sobre el contenido de un flujo, incluyendo **FileStream** y **MemoryStream**. Esta situación parece perfecta para genéricos, pero necesitas tener un medio que de alguna manera te asegure que sólo se utilicen tipos de flujo como argumentos de tipo. No querrás permitir el acceso a un argumento de tipo **int**, por ejemplo. También necesitas un medio para indicarle al compilador que los métodos definidos por el flujo de datos estarán disponibles para su uso. Por ejemplo, tu código genérico necesita un medio para saber que puede invocar al método **Read()**.

Para manejar tal situación, C# proporciona *tipos limitados*. Cuando especificas un parámetro de tipo, puedes especificar también una limitación que el primero debe satisfacer. Esto se realiza utilizando la cláusula **where** cuando se especifica el parámetro de tipo, como se muestra a continuación:

```
class nombre-clase<parámetro-de-tipo> where parámetro-de-tipo : limitaciones { // ...}
```

Aquí, *limitaciones* es una lista de limitaciones separada por comas.

C# define los siguientes tipos de limitaciones:

- 1.** Puedes requerir que cierta clase base esté presente en el argumento de tipo utilizando la *limitación clase base*. Esta limitación se especifica nombrando la clase base deseada. Existe una variante de esta limitación, llamada *limitación de tipo desnudo*, en la cual la clase base se especifica como un parámetro de tipo en lugar de hacerlo como un tipo verdadero. Esto te permite establecer una relación entre dos parámetros de tipo.
- 2.** Puedes requerir que una o más interfaces sean implementadas por un argumento de tipo utilizando una *limitación de interfaz*. Esta limitación se realiza especificando el nombre de la interfaz deseada.
- 3.** Puedes requerir que un argumento de tipo aporte un constructor sin parámetros. A esto se le llama una *limitación de constructor*. Se especifica por **new()**.
- 4.** Puedes especificar que un argumento de tipo deba ser un tipo referencia especificando la *limitación tipo referencia: class*.
- 5.** Puedes especificar que el argumento de tipo sea un tipo valor especificando la *limitación tipo valor: struct*.

De todas estas limitaciones, probablemente las de mayor uso sean la limitación clase base y la limitación de interfaz, pero todas ellas son importantes. Cada una de estas limitaciones se examina en las siguientes secciones.

## Utilizar una limitación de clase base

La limitación de clase base te permite especificar una clase base que un argumento de tipo debe heredar. La limitación de clase base sirve para dos propósitos fundamentales. Primero, te permite utilizar los miembros de la clase base especificada por la limitación dentro de una clase genérica. Por ejemplo, puedes invocar un método o utilizar una propiedad de la clase base. Sin la limitación de la clase base, el compilador no tiene manera de saber qué tipo de miembros podría tener un argumento de tipo. Pero proporcionando una limitación de clase base, estás dejando que el compilador sepa que todos los argumentos de tipo tendrán los miembros definidos por la clase base limitada.

El segundo propósito de una clase base limitada es asegurar que sólo se puedan utilizar argumentos de tipo que soportan la base especificada. Esto significa que por cada clase base limitada, el argumento de tipo debe ser la base en sí o una derivada de la misma. Si intentas utilizar un argumento de tipo que no coincida o herede la clase base especificada, dará como resultado un error en tiempo de compilación.

La limitación de clase base utiliza este formato de la cláusula **where**:

where *T* : *nombre-clase-base*

Aquí, *T* es el nombre del parámetro de tipo y *nombre-clase-base* es el nombre de la base. Sólo se puede especificar una clase base.

A continuación presentamos un ejemplo que muestra el mecanismo de la limitación de la clase base. Crea una clase base llamada **MiMetStr**, que define un método público llamado **ReversaStr()** que regresa una versión inversa de su argumento de cadena de caracteres. Por lo mismo, cualquier clase que herede **MiMetStr** tendrá acceso a este método.

```
// Una demostración sencilla de una clase base limitada.
using System;

class MiMetStr {

    // Invierte una cadena de caracteres y presenta el resultado.
    public string ReversaStr(string str) {
        string result = "";

        foreach(char ch in str)
            result = ch + result;

        return result;
    }

    // ...
}

// Clase MiClase hereda MiMetStr.
class MiClase : MiMetStr { }

// Clase MiClase2 no hereda MiMetStr.
class MiClase2 { }
```

```

// Debido a la limitación de la clase base, todos los argumentos de tipo
// especificados por Test deben tener MiMetStr como una clase base.
class Test<T> where T : MiMetStr { ← Utiliza una clase base limitada.

    T obj;

    public Test(T o) {
        obj = o;
    }

    public void MuestraReversa(string str) {
        // OK invocar ReversaStr() sobre obj porque está declarado por
        // la clase base MiMetStr.
        string revStr = obj.ReversaStr(str);
        Console.WriteLine(revStr);
    }
}

class ClaseBaseLimitDemo {
    static void Main() {
        MiMetStr objA = new MiMetStr();
        MiClase objB = new MiClase();
        MiClase2 objC = new MiClase2();

        // Lo siguiente es válido porque MiMetStr es
        // la clase base especificada.
        Test<MiMetStr> t1 = new Test<MiMetStr>(objA);

        t1.MuestraReversa("Esto es una prueba.");

        // Lo siguiente es válido porque MiClase hereda MiMetStr.
        Test<MiClase> t2 = new Test<MiClase>(objB); ← Correcto porque MiClase
        // hereda MiMetStr.

        t2.MuestraReversa("Otra prueba.");

        // Lo siguiente es INVÁLIDO porque MiClase2 NO HEREDA
        // MiMetStr.
        // Test<MiClase2> t3 = new Test<MiClase2>(objC); // ¡Error!
        // t3.MuestraReversa("¡Error!"); ↑
    }
}

```

↑  
Esto no funcionará porque **MiClase2**  
no hereda **MiMetStr**.

En este programa, la clase **MiMetStr** es heredada por **MiClase**, pero no por **MiClase2**. Como se mencionó, **MiMetStr** declara un método llamado **ReversaStr()**, que invierte una cadena de caracteres y regresa el resultado. A continuación, observa que **Test** es una clase genérica que se declara así:

```
class Test<T> where T : MiMetStr {
```

La cláusula **where** estipula que cualquier argumento de tipo especificado por **T** debe tener **MiMetStr** como clase base.

**Test** define una variable de instancia llamada **obj**, que es de tipo **T**, y un constructor. Éstos se muestran aquí:

```
T obj;  
  
public Test(T o) {  
    obj = o;  
}
```

Como puedes ver, el objeto transmitido a **Test()** se almacena en **obj**.

Ahora observa que **Test** declara el método **MuestraReversa()**, que se presenta a continuación:

```
public void MuestraReversa(string str) {  
    // OK invocar ReversaStr() sobre obj porque está declarado por  
    // la clase base MiMetStr.  
    string revStr = obj.ReversaStr(str);  
    Console.WriteLine(revStr);  
}
```

Este método invoca **ReversaStr()** sobre **obj**, que es un objeto **T**, y luego muestra la cadena de caracteres invertida. El punto principal es que la única razón por la que **ReversaStr()** puede ser invocada es porque la limitación de la clase base requiere que cualquier argumento de tipo atado a **T** heredará **MiMetStr**, que declara **ReversaStr()**. Si no se utilizara la limitación de clase base, el compilador no sabría que un método llamado **ReversaStr()** puede ser invocado sobre un objeto de tipo **T**. Puedes probar esto por ti mismo eliminando la cláusula **where**. El programa ya no compilará porque el método **ReversaStr()** será desconocido.

Además de permitir el acceso a sus miembros, la limitación de la clase base impone que sólo los tipos herederos de la base puedan ser utilizados como argumentos de tipo. Por ello las siguientes líneas fueron eliminadas del programa con signos de comentario:

```
//     Test<MiClase2> t3 = new Test<MiClase2>(objC); // ¡Error!  
//     t3.MuestraReversa("¡Error!");
```

Como **MiClase2** no hereda **MiMetStr**, no puede ser utilizada como tipo de argumento cuando se construye un objeto **Test**. Puedes probarlo eliminando los signos de comentario e intentando recompilar el programa.

Antes de continuar, revisemos los dos efectos de la limitación de la clase base: una limitación de clase base permite que una clase genérica accese los miembros de la base. También asegura que sólo aquellos argumentos de tipo que cumplen por completo con la limitación sean válidos, preservando así la seguridad de tipos.

## Utilizar una limitación para establecer una relación entre dos parámetros de tipo

Existe una variante de la limitación a una clase base que te permite establecer una relación entre dos parámetros de tipo. Por ejemplo, analiza la siguiente declaración de clase genérica:

```
class MiClaseGen<T, V> where V : T {
```

En esta declaración, la cláusula **where** le indica al compilador que el argumento de tipo atado a **V** debe ser idéntico al argumento de tipo atado a **T** o heredado de éste. Si esta relación no está presente, dará como resultado un error en tiempo de compilación. Una limitación que utiliza un parámetro de tipo como el que se acaba de mostrar recibe el nombre de *limitación de tipo desnudo*. El siguiente ejemplo la ilustra:

```
// Crea una relación entre dos parámetros de tipo.
using System;

class A {
    //...
}

class B : A {
    // ...
}

// Aquí, V debe heredar T.
class MiClaseGen<T, V> where V : T { ← Esta limitación requiere que el argumento de tipo
    // ...
}   transmitido a T sea necesariamente una clase
  base del argumento de tipo transmitido a V.

class LimitDesnudaDemo {
    static void Main() {

        // Esta declaración es correcta porque B hereda a A.
        MiClaseGen<A, B> x = new MiClaseGen<A, B>();

        // Esta declaración es errónea porque A no hereda B.
        //    MyGenClass<B, A> y = new MyGenClass<B, A>();
    }
}
```

Primero, observa que la clase **B** hereda a la clase **A**. Después, examina las dos declaraciones **MiClaseGen** en **Main()**. Como lo explican los comentarios, la primera declaración:

```
MiClaseGen<A, B> x = new MiClaseGen<A, B>();
```

es legal porque **B** hereda **A**. Sin embargo, la segunda declaración:

```
//    MyGenClass<B, A> y = new MyGenClass<B, A>();
```

es ilegal porque **A** no hereda **B**.

## Utilizar una interfaz limitada

La limitación de interfaz te permite especificar una interfaz que debe implementar un argumento de tipo. La limitación de interfaz sirve a los mismos dos propósitos que la limitación de clase base. Primero, te permite utilizar los miembros de la interfaz dentro de una clase genérica. Segundo, asegura que sólo se utilicen los argumentos de tipo que implementen la interfaz especificada. Esto

significa que por cualquier limitación de interfaz dada, el argumento de tipo debe ser la interfaz misma o un tipo que implemente la misma.

La limitación de interfaz utiliza el siguiente formato de la cláusula **where**:

where *T* : *nombre-interfaz*

Aquí, *T* es el nombre del parámetro de tipo, y *nombre-interfaz* es el nombre de la interfaz. Se puede especificar más de una interfaz utilizando una lista separada por comas. Si una limitación incluye tanto una clase base como una interfaz, entonces la clase base debe listarse primero.

El siguiente programa ilustra la limitación de interfaz:

```
// Una sencilla demostración de limitación de interfaz.
using System;

// Una interfaz sencilla.
interface IMiInterfaz {
    void Inicio();
    void Fin();
}

// Clase MiClase implementa IMiInterfaz.
class MiClase : IMiInterfaz {
    public void Inicio() {
        Console.WriteLine("Comienza...");
    }
    public void Fin() {
        Console.WriteLine("Finaliza...");
    }
}

// Clase MiClase2 no implementa IMiInterfaz.
class MiClase2 { }

// Debido a la limitación de interfaz, todos los argumentos de tipo
// especificados por Test deben implementar IMiInterfaz.
class Test<T> where T : IMiInterfaz { ← Requiere que todos los argumentos
    T obj;
    de tipo transmitidos a T implementen
    IMiInterfaz.

    public Test(T o) {
        obj = o;
    }

    public void Activa() {
        // OK invocar Inicio() y Fin() porque
        // son declarados por IMiInterfaz.
```

```

    obj.Inicio();
    obj.Fin();
}

class InterfazLimitDemo {
    static void Main() {
        MiClase objA = new MiClase();
        MiClase2 objB = new MiClase2();

        // Lo siguiente es válido porque MiClase implementa IMiInterfaz.
        Test<MiClase> t1 = new Test<MiClase>(objA);

        t1.Activa();

        // Lo siguiente es inválido porque MiClase2 NO IMPLEMENTA
        // IMiInterfaz.
        // Test<MiClase2> t2 = new Test<MiClase2>(objB);
        // t2.Activa();
    }
}

```

Primero, el programa crea una interfaz llamada **IMiInterfaz**, que define dos métodos llamados **Inicio()** y **Fin()**. Luego, el programa define tres clases. La primera de ellas, **MiClase**, implementa **IMiInterfaz**. La segunda, **MiClase2**, no lo hace. La tercera es una clase genérica llamada **Test**. Observa que esta última utiliza una limitación de interfaz para requerir que **T** implemente la interfaz **IMiInterfaz**. También observa que un objeto de tipo **T** es transmitido al constructor de **Test** y almacenado en **obj**. **Test** define un método llamado **Activa()**, que utiliza **obj** para invocar los métodos **Inicio()** y **Fin()** declarados por **IMiInterfaz**.

En **Main()**, el programa crea objetos de **MiClase** y **MiClase2**, llamados **objA** y **objB**, respectivamente. Luego crea un objeto **Test** llamado **t1**, utilizando **MiClase** como argumento de tipo. Esto funciona porque **MiClase** implementa **IMiInterfaz**, con lo que satisface la limitación de interfaz. No obstante, un intento por crear un objeto **Test** llamado **t2** utilizando **MiClase2** como argumento de tipo fracasará porque **MiClase2** no implementa **IMiInterfaz**. Puedes probarlo eliminando los signos de comentario de las últimas dos líneas.

## Utilizar el constructor **new( )** limitado

El constructor **new()** limitado te permite inicializar un objeto de tipo genérico. Por lo regular, no puedes crear una instancia de un parámetro de tipo genérico. Sin embargo, el **new()** limitado cambia la situación porque requiere que un argumento de tipo aporte un constructor sin parámetros. (Este constructor sin parámetros puede ser el constructor por defecto proporcionado automáticamente cuando no se declaran de manera explícita los constructores.) Con el **new()** limitado en posición, puedes invocar el parámetro sin constructor para crear un objeto de tipo genérico.

A continuación presentamos un ejemplo que ilustra el uso de **new()**:

```
// Muestra un constructor new() limitado en acción.  
using System;  
  
class MiClase {  
  
    public MiClase() {  
        Console.WriteLine("Crea una instancia MiClase.");  
        // ...  
    }  
  
    //...  
}  
  
class Test<T> where T : new() { ←———— Requiere que todos los argumentos de  
    T obj;  
  
    public Test() {  
        Console.WriteLine("Crea una instancia T.");  
  
        // Lo siguiente funciona debido a la limitación de new().  
        obj = new T(); // crea un objeto T  
    }  
  
    // ...  
}  
  
class ConsLimitDemo {  
    static void Main() {  
  
        Test<MiClase> t = new Test<MiClase>();  
    }  
}
```

El programa genera los siguientes datos:

```
Crea una instancia T.  
Crea una instancia MiClase.
```

Primero, observa la declaración de la clase **Test**, que se muestra aquí:

```
class Test<T> where T : new() {
```

Debido a la limitación de **new()**, cualquier argumento de tipo debe proporcionar un constructor sin parámetros. Como se explicó, éste puede ser el constructor por defecto o uno que tú crees.

A continuación, examina el constructor **Test**, que se muestra aquí:

```
public Test() {
    Console.WriteLine("Crea una instancia T.");
    // Lo siguiente funciona debido a la limitación de new().
    obj = new T(); // crea un objeto T
}
```

Es creado un nuevo objeto de tipo **T**, y se le asigna a **obj** una referencia al mismo. Esta declaración es válida porque el **new()** limitado asegura que el constructor estará disponible. Para probarlo, intenta eliminar la limitación de **new()** y luego haz el intento por recompilar el programa. Como verás, se reportará un error.

En **Main()**, se inicializa un objeto de tipo **Test**, como se muestra aquí:

```
Test<MiClase> t = new Test<MiClase>();
```

Observa que el argumento de tipo es **MiClase** y que ésta define un constructor sin parámetros. Así, es válido que se utilice como argumento de tipo para **Test**. Debemos destacar que no fue necesario que **MiClase** declarara explícitamente un constructor sin parámetros. Su constructor por defecto también satisface la limitación. No obstante, si una clase necesita otros constructores además de uno sin parámetros, entonces será necesario declarar también y de manera explícita una versión sin parámetros.

Aquí tenemos tres puntos importantes sobre el uso de **new()**. Primero, puede ser utilizado con otras limitaciones, pero debe ser la última limitante en la lista. Segundo, **new()** te permite construir un objeto utilizando sólo el constructor sin parámetros, aun cuando otros constructores estén disponibles. En otras palabras, no está permitido transmitir argumentos a un constructor de un parámetro de tipo. Tercero, no puedes utilizar **new()** junto con una limitación de tipo valor, como se describe a continuación.

## Las limitaciones del tipo referencia y del tipo valor

Las siguientes dos limitaciones te permiten indicar que un argumento de tipo debe ser de tipo valor o de tipo referencia. Éstas son útiles en los pocos casos en los cuales la diferencia entre valor y referencia son de importancia para el código genérico. He aquí el formato general de la limitación del tipo referencia:

where *T* : class

En este formato de la cláusula **where**, la palabra clave **class** especifica que *T* debe ser un tipo referencia. Así, el intento de utilizar un tipo valor, como **int** o **bool**, para *T* resultará en un error de compilación.

A continuación presentamos el formato general de una limitación de tipo valor:

where *T* : struct

En este caso, la palabra clave **struct** especifica que *T* debe ser de tipo valor. (Recuerda que todas las estructuras son tipos valor.) De esta manera, el intento de utilizar un tipo referencia, como **string**, para *T* resultará en un error de compilación. En ambos casos, cuando están presentes limitaciones adicionales, **class** o **struct** deben ser los primeros limitantes en la lista.

He aquí un ejemplo que muestra las limitaciones de los tipos de referencia:

```
// Muestra una limitación de referencia.
using System;

class MiClase {
    //...
}

// Utiliza una limitación de referencia.
class Test<T> where T : class { ← Sólo se pueden transmitir a T tipos referencia.
    T obj;

    public Test() {
        // La siguiente declaración es legal sólo porque
        // se tiene la garantía de que T es un tipo referencia, al cual
        // se le puede asignar un valor null.
        obj = null;
    }

    // ...
}

class ClaseLimitDemo {
    static void Main() {

        // Lo siguiente es correcto porque MiClase es una clase.
        Test<MiClase> x = new Test<MiClase>();

        // La siguiente línea es errónea porque int es un tipo valor.
        // Test<int> y = new Test<int>();
    }
}
```

Primero, observa cómo está declarada **T**:

```
class Test<T> where T : class {
```

La limitación **class** requiere que cualquier argumento de tipo para **T** sea un tipo referencia. En este programa, esto es necesario debido a lo que ocurre dentro del constructor **Test**:

```
public Test() {
    // La siguiente declaración es legal sólo porque
    // se tiene la garantía de que T es un tipo referencia, al cual
    // se le puede asignar un valor null.
    obj = null;
}
```

Aquí, a **obj** (que es de tipo **T**) se le asigna el valor **null**. Esta asignación es válida sólo para tipos referencia. Como regla general, no se le puede asignar **null** a un tipo valor. (La excepción a esta regla es el *tipo anulable*, que es un tipo especial de estructura que encapsula un tipo valor y permite el valor nulo. Ver capítulo 14 para detalles.) Por lo mismo, sin la limitación, la asignación no sería válida y la compilación fallaría. Éste es un caso en el que la diferencia entre tipos valor y tipos referencia puede tener importancia para una rutina genérica.

La limitación para tipo valor es el complemento de la limitación para tipo referencia. Simplemente asegura que cualquier tipo de argumento sea de tipo valor, incluyendo **struct** y **enum**. (En este contexto, un tipo anulable no se considera un tipo valor.) He aquí un ejemplo:

```
// Muestra la limitación de un tipo valor.
using System;

struct MiStruct {
    //...
}

class MiClase {
    // ...
}

class Test<T> where T : struct { ←———— Sólo tipos valor pueden pasar a T.
    T obj;

    public Test(T x) {
        obj = x;
    }

    // ...
}

class ValorLimitDemo {
    static void Main() {

        // Ambas declaraciones son legales.
        Test<MiStruct> x = new Test<MiStruct>(new MiStruct());
        Test<int> y = new Test<int>(10);

        // Pero, la siguiente declaración ¡es ilegal!
        //     Test<MiClase> z = new Test<MiClase>(new MiClase());
    }
}
```

En este programa, **Test** se declara de la forma siguiente:

```
class Test<T> where T : struct {
```

Como **T** de **Test** ahora tiene una limitación **struct**, **T** puede ser atada sólo a argumentos de tipo valor. Esto significa que **Test<MiStruct>** y **Test<int>** son válidos, pero no así **Test<MiClase>**. Para probarlo, intenta eliminar los signos de comentario de la última línea y recompila el programa. Aparecerá un reporte de error.

## Utilizar limitaciones múltiples

Puede existir más de una limitación asociada con un parámetro. Cuando así sucede, se utiliza una lista de limitaciones separadas por comas. En esta lista, la primera limitación debe ser **class** o **struct** (si está presente), o la clase base (si está especificada alguna). Es legal especificar limitantes tanto **class** como **struct** y una limitación de clase base. A continuación en la lista debe seguir una limitación de interfaz. La limitación **new()** debe ir al final. Por ejemplo, la siguiente es una declaración válida:

```
class MiClaseGen<T> where T : MiClase, IMiInterfaz, new() { // ... }
```

En este caso, **T** debe ser reemplazada por un argumento de tipo que hereda **MiClase**, implementa **IMiInterfaz**, y tiene un constructor sin parámetros.

Cuando se utilizan dos o más parámetros, puedes especificar una limitación para cada parámetro utilizando una cláusula **where** por separado. Por ejemplo:

```
// Utiliza múltiples cláusulas where.
using System;

// DosWhere tiene dos argumentos de tipo y ambos tienen una cláusula
// where.
class DosWhere<T, V> where T : class
    where V : struct { ← Dos cláusulas where. Una
        T ob1;
        V ob2;

        public DosWhere(T t, V v) {
            ob1 = t;
            ob2 = v;
        }
    }

class DosWhereDemo {
    static void Main() {
        // Esto es correcto porque string es una clase e int es un tipo
        // valor.
        DosWhere<string, int> obj =
            new DosWhere<string, int>("test", 11);

        // Lo siguiente es erróneo porque bool no es un tipo de referencia.
        // DosWhere<bool, int> obj2 =
        //     new DosWhere<bool, int>(true, 11);
    }
}
```

En este ejemplo, **DosWhere** toma dos argumentos de tipo y ambos tienen una cláusula **where**. Pon especial atención a esta declaración:

```
class DosWhere<T, V> where T : class
    where V : struct {
```

Observa que lo único que separa la primera cláusula **where** de la segunda es un espacio en blanco. Ninguna otra puntuación es requerida ni válida.

## Crear un valor por omisión de un parámetro de tipo

Cuando se escribe código genérico, habrá ocasiones en las que la diferencia entre tipos valor y tipos parámetro sea de gran importancia. Una de tales situaciones ocurre cuando quieras dar un valor por defecto a una variable de tipo parámetro. Para los tipos referencia, el valor por defecto es **null**. Para los tipos diferentes a **struct** el valor por defecto es 0. El valor por defecto para un **struct** es un objeto de esa misma estructura con todos sus campos establecidos en su valor por defecto. Así, el problema surge si quieres darle un valor por defecto a una variable de tipo parámetro. ¿Qué valor utilizaría: **null**, 0 o algo más?

Por ejemplo, dada una clase genérica llamada **Test** declarada de la siguiente manera:

```
class Test<T> {
    T obj;
    // ...
```

Si quieres dar a **obj** un valor por defecto, utilizarías

```
obj = null; // funciona sólo para tipos referencia
u
obj = 0 // funciona sólo con tipos numéricos y enum, pero no con
estructuras
```

La solución a este problema es utilizar otro formato de **default**, como se muestra a continuación:

**default(tipo)**

Éste es el formato operacional de **default**, y produce un valor por defecto del *tipo* especificado, no importa qué tipo se utilice. Así, para continuar con el ejemplo, para asignar a **obj** un valor por defecto de tipo **T**, utilizarías esta declaración:

```
obj = default(T);
```

Esto funciona con cualquier tipo de argumento, ya sea de tipo valor o referencia.

A continuación mostramos un programa breve que muestra el uso de **default**:

```
// Muestra el operador default en acción.
using System;
class MiClase {
```

```

    //...
}

// Construye un valor por defecto de T.
class Test<T> {
    public T obj;

    public Test() {
        // La siguiente declaración funcionará solamente para tipos
        // referencia.
        //    obj = null; // no se puede utilizar

        // La siguiente declaración funcionará solamente para tipos valor.
        //    obj = 0; // no se puede utilizar

        // Esta declaración funciona tanto para los tipos referencia como los
        // tipos valor.
        obj = default(T); ← Crea un valor por defecto para cualquier T.
    }
}

class DefaultDemo {
    static void Main() {

        // Construye Test utilizando un tipo referencia.
        Test<MiClase> x = new Test<MiClase>();
    }
}

```

## Pregunta al experto

**P:** Algunas de las declaraciones relacionadas con tipos genéricos son *muy largas*. ¿Existe alguna manera de acortarlas?

**R:** Las nuevas características de C# 3.0 sobre variables de tipo implícito pueden acortar una declaración larga que incluye un inicializador. Como lo sabes desde el capítulo 2, en una declaración **var**, el tipo de la variable es determinado por el tipo del inicializador. Por lo mismo, una declaración como

```
UnaClase<String, bool> UnObj =
    new UnaClase<String, bool>("prueba", false);
```

Puede ser escrita de manera compacta como

```
var UnObj = new UnaClase<String, bool>("prueba", false);
```

Aunque el uso de **var**, en este caso, acorta la declaración, su uso principal es en tipos anónimos, que se describen en el capítulo 14. Más aún, dado que las variables de tipo implícito son nuevas para C#, no está claro (en el momento en que se escribe este libro) que el anterior uso de **var** pueda ser considerado una de las “mejores prácticas” por los programadores de C#. Además, los estándares de código en tu lugar de trabajo pueden excluir este uso.

```

if(x.obj == null)
    Console.WriteLine("x.obj es null.");

// Construye Test utilizando un tipo valor.
Test<int> y = new Test<int>();

if(y.obj == 0)
    Console.WriteLine("y.obj es 0.");
}
}

```

Los datos generados por este programa son:

```

x.obj es null.
y.obj es 0.

```

## Estructuras genéricas

Puedes crear una estructura que tome parámetros de tipo. La sintaxis para una estructura genérica es la misma que para las clases genéricas. Por ejemplo, en el siguiente programa, la estructura **ClaveValor**, que almacena pares clave/valor, es genérica:

```

// Muestra una estructura genérica.
using System;

// Esta estructura es genérica.
struct ClaveValor<TClave, TValue>{ ← Una estructura genérica.
    public TClave key;
    public TValue val;

    public ClaveValor(TClave a, TValue b) {
        key = a;
        val = b;
    }
}

class GenStructDemo {
    static void Main() {
        ClaveValor<string, int> kv =
            new ClaveValor<string, int>("Tom", 20);

        ClaveValor<string, bool> kv2 =
            new ClaveValor<string, bool>("Ventilador Encendido", false);

        Console.WriteLine(kv.key + " tiene " + kv.val + " años.");
        Console.WriteLine(kv2.key + " es " + kv2.val);
    }
}

```

Los datos generados por este programa son:

```
Tom tiene 20 años.  
Ventilador Encendido es False
```

Como las clases genéricas, las estructuras genéricas pueden tener limitaciones. Por ejemplo, esta versión de **ClaveValor** limita **TValor** a tipos de valor:

```
struct ClaveValor<TClave, TValor> where TValor : struct {  
    // ...
```

## Métodos genéricos

Como lo han mostrado los ejemplos anteriores, los métodos dentro de una clase genérica pueden hacer uso de un parámetro de tipo de una clase y, por tanto, ser automáticamente un genérico relativo a ese parámetro de tipo. Sin embargo, es posible declarar un método genérico que utilice uno o más parámetros de tipo de sí mismo. Más aún, es posible crear un método genérico que se encuentre contenido dentro de una clase no genérica.

Comencemos con un ejemplo. El siguiente programa declara una clase no genérica llamada **ArregloUtils**, un método estático genérico llamado **CopiaPega()** que se localiza dentro de la mencionada clase. El método **CopiaPega()** copia el contenido de un arreglo a otro, insertando un nuevo elemento en un lugar especificado durante el proceso. Puede utilizarse para cualquier tipo de arreglo.

```
// Muestra un método genérico en acción.  
using System;  
  
// Una clase de utilidades de arreglo. Observa que no se trata de  
// una clase genérica.  
class ArregloUtils {  
  
    // Copia un arreglo, inserta un nuevo elemento en el proceso.  
    // Éste es un método genérico.  
    public static bool CopiaPega<T>(T e, int idx, ←———— Un método genérico.  
   T[] fuente, T[] objetivo) {  
  
        // Verifica si objetivo es lo suficientemente grande.  
        if(objetivo.Length < fuente.Length+1)  
            return false;  
  
        // Copias fuente a objetivo, inserta e en idx en el proceso.  
        for(int i=0, j=0; i < fuente.Length; i++, j++) {  
            if(i == idx) {  
                objetivo[j] = e;  
                j++;  
            }  
            objetivo[j] = fuente[i];  
        }  
    }  
}
```

```

        return true;
    }
    // ...
}

class GenMetDemo {
    static void Main() {
        int[] nums = { 1, 2, 3 };
        int[] nums2 = new int[4];

        // Muestra el contenido de nums.
        Console.Write("Contenido de nums: ");
        foreach (int x in nums)
            Console.Write(x + " ");

        Console.WriteLine();

        // Opera sobre un arreglo int.
        ArregloUtils.CopiaPega(99, 2, nums, nums2); ← Invoca el método genérico. El tipo de T es determinado por la interfaz. En este caso, el tipo es int.

        // Muestra el contenido de nums2.
        Console.Write("Contenido de nums2: ");
        foreach (int x in nums2)
            Console.Write(x + " ");

        Console.WriteLine();

        // Ahora, usa CopiaPega sobre un arreglo string.
        string[] strs = { "Los Genéricos", "son", "poderosos." };
        string[] strs2 = new string[4];

        // Muestra el contenido de strs.
        Console.Write("Contenido de strs: ");
        foreach(string s in strs)
            Console.Write(s + " ");

        Console.WriteLine();

        // Inserta en un arreglo string.
        ArregloUtils.CopiaPega("en C#", 1, strs, strs2); ← El tipo T aquí es string.

        // Muestra el contenido de strs2.
        Console.Write("Contenido de strs2: ");
        foreach (string s in strs2)
            Console.Write(s + " ");

        Console.WriteLine();
    }
}

```

```
// Esta invocación es inválida porque el primer argumento
// es de tipo double, y el tercero y cuarto argumentos
// tienen tipos de elemento int.
//     ArregloUtils.CopiaPega(0.01, 2, nums, nums2);
}
}
```

Los datos generados por el programa son:

```
Contenido de nums: 1 2 3
Contenido de nums2: 1 2 99 3
Contenido de strs: Los Genéricos son poderosos.
Contenido de strs2: Los Genéricos en C# son poderosos.
```

Examinemos **CopiaPega()** de cerca. Primero, observa cómo se declara en esta línea:

```
public static bool CopiaPega<T>(T e, int idx,
                                  T[] fuente, T[] objetivo) {
```

Los parámetros de tipo son declarados *después* del nombre del método, pero antes de la lista de parámetros. También advierte que **CopiaPega()** es estático, permitiéndole que sea invocado independientemente de cualquier objeto. Debes entender, sin embargo, que los métodos genéricos pueden ser estáticos o no estáticos. No existe restricción al respecto.

Ahora, observa cómo es invocado **CopiaPega()** dentro de **Main()** utilizando una sintaxis normal de invocación, sin necesidad de invocar argumentos de tipo. Esto se debe a que los tipos de los argumentos de tipo son discernidos automáticamente con base en el tipo de dato utilizado para invocar **CopiaPega()**. Con base en esta información, el tipo **T** se ajusta en consecuencia. Este proceso recibe el nombre de *inferencia de tipo*. Por ejemplo, en la primera invocación:

```
ArregloUtils.CopiaPega(99, 2, nums, nums2);
```

el tipo de **T** se transforma en **int** porque 99 es un entero, y los tipos de elemento de **nums** y **nums2** son **int**. En la segunda invocación, se utilizan cadenas de texto (tipo **string**), y **T** es reemplazado por **string**.

Ahora, observa el código comentado:

```
//     ArregloUtils.CopiaPega(0.01, 2, nums, nums2);
```

Si eliminas los signos de comentarios e intentas recompilar el programa, recibirás un mensaje de error. La razón es que el tipo del primer argumento es **double**, pero los tipos de elementos de **nums** y **nums2** son **int**. Sin embargo, los tres tipos deben ser sustituidos por el mismo parámetro de tipo, **T**. Esto provoca una discordancia de tipo, que resulta en un error en tiempo de compilación. Esta capacidad de cuidar la seguridad de tipo es una de las ventajas más importantes de los métodos genéricos.

La sintaxis utilizada para crear **CopiaPega()** se puede generalizar. He aquí el formato general de un método genérico:

*tipo-ret nombre-met<lista-parámetros-de-tipo>(lista-parámetros) { //...*

En todos los casos, *lista-parámetros-de-tipo* es una lista de los parámetros de tipo separados por comas. Observa que para un método genérico, la lista de parámetros de tipo es posterior al nombre del método.

## Utilizar argumentos de tipo explícito para invocar un método genérico

Aunque la interfaz de tipo implícito es adecuada para la mayoría de las invocaciones de un método genérico, es posible especificar explícitamente el argumento de tipo. Para hacerlo, especifica el argumento de tipo después del nombre del método cuando invoques este último. Por ejemplo, aquí **CopiaPega()** se especifica de manera explícita como tipo **string**:

```
ArregloUtils.CopiaPega<string>("en C#", 1, strs, strs2);
```

Necesitarás especificar explícitamente el tipo cuando el compilador no pueda inferir el tipo de un parámetro de tipo.

## Utilizar una limitación con un método genérico

Puedes añadir limitaciones a los argumentos de tipo de un método genérico especificándolas después de la lista de parámetros. Por ejemplo, la siguiente versión de **CopiaPega()** funcionará sólo con tipos de referencia:

```
public static bool CopiaPega<T>(T e, int idx,
                                  T[] fuente, T[] destino) where T : class {
```

Si intentaras ejecutar esta versión en el programa mostrado anteriormente, la siguiente invocación a **CopiaPega()** no compilaría porque **int** es un tipo valor, no un tipo referencia:

```
// Ahora es erróneo ¡porque T debe ser un tipo referencia!
ArregloUtils.CopiaPega(99, 2, nums, nums2); // ¡Ahora es ilegal!
```

## Pregunta al experto

**P:** Usted mencionó que hay algunos casos en los cuales el compilador no puede inferir el tipo que debe utilizar para un parámetro de tipo cuando se invoca un método genérico y que el tipo necesita ser especificado explícitamente. ¿Puede dar un ejemplo?

**R:** Sí. Entre otros, esta situación ocurre cuando un método genérico no tiene parámetros. Por ejemplo, analiza este método genérico:

```
class UnaClase {
    public static T UnMétodo<T>() where T : new() {
        return new T();
    }
    // ...
```

(continúa)

Cuando se invoca este método, no hay argumentos a partir de los cuales el tipo de **T** pueda ser inferido. El tipo de regreso de **T** no es suficiente para que se realice la inferencia. Por lo mismo, esto no funcionará:

```
unObj = UnaClase.UnMétodo(); // no funcionará
```

En lugar de eso, debe ser invocado con una especificación explícita de tipo. Por ejemplo:

```
unObj = UnaClase.UnMétodo<MiClase>; // arreglado
```

## Delegados genéricos

Como los métodos, los delegados también pueden ser genéricos. Para declarar un delegado genérico, utiliza este formato general:

```
delegate tipo-ret nombre-delegado<lista-parámetros-de-tipo >(lista-argumentos);
```

Observa el lugar que ocupa la lista de parámetros de tipo. Sigue inmediatamente después del nombre del delegado. La ventaja de los delegados genéricos es que te permiten definir, con seguridad de tipo, un formato generalizado que luego puede coincidir con cualquier método compatible.

El siguiente programa muestra un delegado genérico llamado **Invierte** que tiene un parámetro de tipo llamado **T**. Regresa un tipo **T** y toma un argumento de tipo **T**.

```
// Muestra un delegado genérico en acción.
using System;

// Declara un delegado genérico.
delegate T Invierte<T>(T v); ←———— Un delegado genérico.

class GenDelegadoDemo {

    // Regresa el recíproco de un double.
    static double Recip(double v) {
        return 1 / v;
    }

    // Invierte una cadena de caracteres y regresa el resultado.
    static string ReversaStr(string str) {
        string result = "";

        foreach (char ch in str)
            result = ch + result;

        return result;
    }
}
```

```

static void Main() {
    // Construye dos delegados Invierte.
    Invierte<double> invDel = Recip;
    Invierte<string> invDel2 = ReversaStr; ← Crea instancias double
  y string de Invierte.
    Console.WriteLine("El recíproco de 4 es " + invDel(4.0));
    Console.WriteLine();
    string str = "ABCDEFG";
    Console.WriteLine("Cadena original: " + str);
    str = invDel2(str);
    Console.WriteLine("Cadena invertida: " + str);
}
}

```

Los datos generados por el programa son:

El recíproco de 4 es 0.25

Cadena original: ABCDEFG  
Cadena invertida: GFEDCBA

Veamos de cerca este programa. Primero, observa cómo se declara el delegado **Invierte**:

```
delegate T Invierte<T>(T v);
```

Observa que **T** puede ser utilizada como el tipo de regreso, aunque el parámetro de tipo **T** es especificado después del nombre **Invierte**.

Dentro de **GenDelegadoDemo**, se declaran los métodos **Recip()** y **ReversaStr()**, como se muestra aquí:

```

static double Recip(double v) {
    static string ReversaStr(string str) {

```

El método **Recip()** regresa el recíproco del valor **double** transmitido como argumento. El método **ReversaStr()**, que está adaptado de un ejemplo anterior, invierte una cadena de caracteres y regresa el resultado.

Dentro de **Main()**, un delegado llamado **invDel** es convertido en instancia y se le asigna una referencia para **Recip()**.

```
Invierte<double> invDel = Recip;
```

Dado que **Recip()** toma un argumento **double** y regresa un valor **double**, **Recip()** es compatible con la instancia **double** de **Invierte**.

De manera similar, se crea el delegado **invDel2** y se le asigna una referencia a **ReversaStr()**.

```
Invierte<string> invDel2 = ReversaStr;
```

Dado que **ReversaStr()** toma un argumento de cadena de texto y regresa un resultado de cadena, es compatible con la versión string de **Invierte**.

Como la seguridad de tipos se hereda en los genéricos, no puedes asignar métodos incompatibles a los delegados. Por ejemplo, suponiendo la ejecución del programa anterior, la siguiente declaración sería un error:

```
Invierte<int> invDel = ReversaStr; // ¡Error!
```

Dado que **ReversaStr()** toma como argumento una cadena de caracteres y regresa como resultado una cadena de caracteres, no se le puede asignar una versión **int** de **Invierte**.

## Interfaces genéricas

Las interfaces genéricas se especifican como las clases genéricas. Aquí tenemos un ejemplo. Crea una interfaz genérica llamada **IDosDCoord** que define métodos que obtienen y establecen valores para las coordenadas X y Y. Por lo mismo, cualquier clase que implemente esta interfaz soportará las coordenadas X y Y. El tipo de datos de las coordenadas se especifica por el parámetro de tipo. Luego, **IDosDCoord** es implementada por dos clases diferentes.

```
// Muestra una interfaz genérica.
using System;

// Esta interfaz es genérica. Define métodos que soportan
// coordenadas bidimensionales.
public interface IDosDCoord<T> { ← Una interfaz genérica.

    T GetX();
    void SetX(T x);

    T GetY();
    void SetY(T y);
}

// Una clase que encapsula coordenadas bidimensionales.
class XYCoord<T> : IDosDCoord<T>{ ← Implementa una interfaz genérica.
    T X;
    T Y;

    public XYCoord(T x, T y) {
        X = x;
        Y = y;
    }

    public T GetX() { return X; }
    public void SetX(T x) { X = x; }

    public T GetY() { return Y; }
    public void SetY(T y) { Y = y; }
}
```

```
// Una clase que encapsula coordenadas tridimensionales.
class XYZCoord<T> : IDosDCoord<T> { ← Implementa una interfaz genérica.
    T X;
    T Y;
    T Z;

    public XYZCoord(T x, T y, T z) {
        X = x;
        Y = y;
        Z = z;
    }

    public T GetX() { return X; }
    public void SetX(T x) { X = x; }

    public T GetY() { return Y; }
    public void SetY(T y) { Y = y; }

    public T GetZ() { return Z; }
    public void SetZ(T z) { Z = z; }
}

class GenInterfazDemo {

    // Un método genérico que puede mostrar las coordenadas X,Y asociadas
    // con cualquier objeto que implemente la interfaz genérica IDosDCoord.
    static void ShowXY<T>(IDosDCoord<T> xy) {
        Utiliza métodos
        Console.WriteLine(xy.GetX() + ", " + xy.GetY()); ← especificados por
    }  IDosDCoord.

    static void Main() {

        XYCoord<int> xyObj = new XYCoord<int>(10, 20);
        Console.Write("Los valores X,Y en xyObj: ");
        ShowXY(xyObj);

        XYZCoord<double> xyzObj = new XYZCoord<double>(-1.1, 2.2, 3.1416);
        Console.Write("El componente X,Y de xyzObj: ");
        ShowXY(xyzObj);
    }
}
```

Los datos generados por el programa son:

```
Los valores X,Y en xyObj: 10, 10
El componente X,Y de xyzObj: -1.1, 2.2
```

Existen varias cosas de interés en el ejemplo anterior. Primero, cómo se declara **IDosDCoord**:

```
public interface IDosDCoord<T> {
```

Como se mencionó, una interfaz genérica utiliza una sintaxis similar a la de las clases genéricas.

Ahora, observa cómo se declara **XYCoord**, que implementa **IDosDCoord**:

```
class XYCoord<T> : IDosDCoord<T> {
```

El parámetro de tipo **T** se especifica por **XYCoord** y también se especifica en **IDosDCoord**. Esto es importante. Una clase que implementa una versión genérica de una interfaz también genérica debe, en sí, ser genérica. Por ejemplo, la siguiente declaración sería ilegal porque **T** no está definida:

```
class XYCoord : IDosDCoord<T> { // ;Error!
```

El parámetro de tipo requerido por **IDosDCoord** debe ser especificado por la clase que lo implementa, que en este caso es **XYCoord**. De lo contrario, no hay manera de que la interfaz reciba el argumento de tipo.

A continuación, **XYCoord** declara dos variables llamadas **X** y **Y** que contienen las coordenadas. Éstas son, como era de esperarse, objetos de tipo genérico **T**. Finalmente, los métodos definidos por **IDosDCoord** se implementan.

**IDosDCoord** también es implementada por la clase **XYZCoord**, que encapsula coordenadas tridimensionales (**X,Y,Z**). Esta clase implementa los métodos definidos por **IDosDCoord** y añade métodos para accesar la coordenada **Z**.

En **GenInterfazDemo**, se define un método genérico llamado **ShowXY()**. Muestra las coordenadas **X,Y** del objeto que le es transmitido. Observa que el tipo de este parámetro es **IDosDCoord**. Esto significa que puede operar sobre cualquier objeto que implementa la interfaz **IDosDCoord**. En este caso, significa que pueden utilizarse como argumentos objetos del tipo **XYCoord** y **XYZCoord**. Este hecho se ilustra con **Main()**.

Un parámetro de tipo para una interfaz genérica puede tener limitantes de la misma manera que para una clase genérica. Por ejemplo, esta versión de **IDosDCoord** restringe su uso a tipos valor:

```
public interface IDosDCoord<T> where T : struct {
```

Cuando se implementa esta versión, la clase implementada también debe especificar la misma limitación para **T**, como se muestra aquí:

```
class XYCoord<T> : IDosDCoord<T> where T : struct {
```

Dada la limitación del tipo valor, esta versión de **XYCoord** no puede utilizarse en tipos clase, por ejemplo. Así, la siguiente declaración sería desechada:

```
// Ahora, esto no funcionará.  
XYCoord<string> xyObj = new XYCoord<string>("10", "20");
```

Dado que **string** no es un tipo valor, su uso con **XYCoord** es ilegal.

Aunque una clase que implementa una versión genérica de una interfaz genérica debe, en sí, ser genérica, como se explicó anteriormente, una clase no genérica *puede* implementar una versión específica de una interfaz genérica. Por ejemplo, aquí, **XYCoordInt** implementa explícitamente **IDosDCoord<int>**:

```
class XYCoordInt : IDosDCoord<int> {
    int X;
    int Y;

    public XYCoordInt(int x, int y) {
        X = x;
        Y = y;
    }

    public int GetX() { return X; }
    public void SetX(int x) { X = x; }

    public int GetY() { return Y; }
    public void SetY(int y) { Y = y; }
}
```

Observa que **IDosDCoord** es especificado con un tipo **int** explícito. Por lo mismo, **XYCoordInt** no necesita tomar un argumento de tipo porque no lo transmite junto con **IDosDCoord**.

Otro punto: aunque la declaración de una propiedad no puede, por sí misma, especificar un parámetro de tipo, una propiedad declarada en una clase genérica puede utilizar un parámetro de tipo que sea declarado por la clase genérica. Por lo mismo, los métodos **GetX()**, **GetY()** y demás del anterior programa pueden ser convertidos en propiedades que utilicen el parámetro de tipo **T**. Esta tarea se le deja al lector como ejercicio en la sección Autoexamen al final de este capítulo.

## Pregunta al experto

**P:** ¿Puedo comparar dos instancias de un parámetro de tipo utilizando alguno de los operadores == o != ?

**R:** La respuesta tiene dos partes: Primera, si el parámetro de tipo especifica una referencia o una limitación de clase base, entonces == y != están permitidos, pero sólo se utilizan para probar igualdad de referencia. Por ejemplo, el siguiente método no compilará:

```
public static bool MismoValor<T>(T a, T b) {
    if(a == b) return true; // no funcionará
    return false;
}
```

Dado que **T** es un tipo genérico, el compilador no tiene manera de saber con precisión cómo debe comparar ambos objetos buscando igualdad. ¿Debe utilizarse una comparación bitwise? ¿Sólo deben compararse ciertos campos? ¿Se debe utilizar equidad de referencia? El compilador no tiene manera de conocer esas respuestas.

(continúa)

A primera vista, parece ser un problema serio. Por fortuna no es tal, porque C# proporciona un mecanismo a través del cual puedes determinar si dos instancias con parámetro de tipo son las mismas. Para permitir que dos objetos con parámetro de tipo genérico sean comparados, utiliza el método **CompareTo()** definido por una de las interfaces estándar: **IComparable**. Esta interfaz tiene formatos tanto genérico como no genérico. **IComparable** es implementada por todos los tipos integrados de C#, incluyendo **int**, **string** y **double**. Es fácil de implementar en las clases que tú mismo creas.

La interfaz **IComparable** define sólo el método **CompareTo()**. Su formato genérico se muestra a continuación:

```
int CompareTo(T obj)
```

Compara el objeto invocado con *obj*. Regresa cero si ambos objetos son iguales, un valor positivo si el objeto invocado es mayor que *obj*, y un valor negativo si el objeto invocado es menor que *obj*.

Para utilizar **CompareTo()**, debes especificar una limitación que requiera que cada argumento de tipo implemente la interfaz **IComparable**. Luego, cuando necesitas comparar dos objetos con parámetro de tipo, simplemente invoca **CompareTo()**. Por ejemplo, he aquí una versión corregida de **MismoValor()**:

```
// Requiere interface IComparable.
public static bool MismoValor<T>(T a, T b) where T : IComparable<T> {
    if(a.CompareTo(b) == 0) return true; // arreglado
    return false;
}
```

Dado que la limitación de interface requiere que **T** implemente **IComparable<T>**, el método **CompareTo()** puede utilizarse para determinar la igualdad. Por supuesto, esto significa que las únicas instancias de clase que implementa **IComparable<T>** pueden ser transmitidas a **MismoValor()**.

## Prueba esto

## Crear un orden en cola genérico

En capítulos anteriores creaste una clase enriquecida para un orden en cola. En esos capítulos, el tipo de datos operados por el programa estaban codificados como **char**. A través del uso de genéricos, puedes convertir fácilmente la clase de orden en cola en un formato que puede operar sobre cualesquier tipos de datos. Ése es el tema de este ejemplo. Dado que el código de orden en cola ha sido modificado y enriquecido a través de varios capítulos, por claridad todas las piezas del código serán mostradas aquí, incluso aquellas que no son afectadas por la transformación a genérico.

## Paso a paso

1. Crea una interfaz genérica llamada **IC** que tome un parámetro de tipo que especifique el tipo de dato que almacenará la cola. A continuación se muestra:

```

// Una interfaz genérica para un orden en cola.
// Aquí, T especifica el tipo de dato almacenado en la cola.
public interface IC<T> {
    // Coloca un objeto en la cola.
    void Put(T obj);

    // Obtiene un objeto de la cola.
    T Get();
}

```

Esta interfaz es similar a **ICharQ** desarrollada en la sección *Prueba esto: Crear una interfaz de orden en cola* del capítulo 9, pero ésta es genérica.

- 2.** Incluye las clases de excepción desarrolladas en *Prueba esto: Crear una interfaz de orden en cola* del capítulo 10. No son afectadas por el cambio a genérico. A continuación se muestran:

```

// Una excepción para errores en cola llena.
class ColaLLenaException : Exception {
    public ColaLLenaException(string str) : base(str) { }
    // Añade aquí otro constructor ColaLLenaException si lo deseas.

    public override string ToString() {
        return "\n" + Message;
    }
}

// Una excepción para errores en cola vacía.
class ColaVaciaException : Exception {
    public ColaVaciaException(string str) : base(str) { }
    // Añade aquí otro constructor ColaVaciaException si lo deseas.

    public override string ToString() {
        return "\n" + Message;
    }
}

```

- 3.** Modifica las clases de cola de manera que implementen la interfaz **IC**. Para ser breves, solamente actualizaremos la clase **ColaSimple**, pero no debes tener problema actualizando las otras dos implementaciones. La versión genérica de **ColaSimple** se muestra a continuación:

```

// Una clase genérica para tamaño fijo de cola.
// Esta clase implementa una interfaz E/S genérica.
class ColaSimple<T> : IC<T> {
    T[] q; // este arreglo contiene la cola
    int putloc, getloc; // los índices put y get

    // Construye una cola vacía dado su tamaño.
    public ColaSimple(int tamaño) {
        q = new T[tamaño+1]; // reserva memoria para la cola
        putloc = getloc = 0;
    }
}

```

(continúa)

```
// Coloca un elemento en la cola.
public void Put(T obj) {
    if(putloc==q.Length-1)
        throw new ColaLlenaException("¡Cola llena! Máxima longitud es " +
                                      (q.Length-1) + ".");
    putloc++;
    q[putloc] = obj;
}

// Obtiene un elemento de la cola.
public T Get() {
    if(getloc == putloc)
        throw new ColaVaciaException("La cola está vacía.");

    getloc++;
    return q[getloc];
}
```

Como puedes ver, el tipo de datos almacenados en la cola se especifican por el parámetro de tipo **T**. Eso significa que **ColaSimple** puede utilizarse para almacenar cualesquier tipos de datos.

4. El siguiente programa ensambla todas las piezas y muestra la cola genérica en acción:

```
// Una clase genérica de orden en cola.

using System;

// Una interfaz genérica para un orden en cola.
// Aquí, T especifica el tipo de dato almacenado en la cola.
public interface IC<T> {
    // Coloca un objeto en la cola.
    void Put(T obj);

    // Obtiene un objeto de la cola.
    T Get();
}

// Una excepción para errores de cola llena.
class ColaLlenaException : Exception {
    public ColaLlenaException(string str) : base(str) { }
    // Añade aquí otro constructor ColaLlenaException si lo deseas.

    public override string ToString() {
        return "\n" + Message;
    }
}
```

```
// Una excepción para errores de cola vacía.  
class ColaVaciaException : Exception {  
    public ColaVaciaException(string str) : base(str) {}  
    // Añade aquí otro constructor ColaVaciaException si lo deseas.  
  
    public override string ToString() {  
        return "\n" + Message;  
    }  
}  
  
// Una clase genérica para tamaño fijo de cola.  
// Esta clase implementa una interfaz E/S genérica.  
class ColaSimple<T> : IC<T> {  
    T[] q; // este arreglo contiene la cola  
    int putloc, getloc; // los índices put y get  
  
    // Construye una cola vacía dado su tamaño.  
    public ColaSimple(int tamaño) {  
        q = new T[tamaño+1]; // reserva memoria para la cola  
        putloc = getloc = 0;  
    }  
  
    // Coloca un elemento en la cola.  
    public void Put(T obj) {  
        if(putloc==q.Length-1)  
            throw new ColaLlenaException("Cola llena! Máxima longitud es " +  
   (q.Length-1) + ".");  
  
        putloc++;  
        q[putloc] = obj;  
    }  
  
    // Obtiene un elemento de la cola.  
    public T Get() {  
        if(getloc == putloc)  
            throw new ColaVaciaException("La cola está vacía.");  
  
        getloc++;  
        return q[getloc];  
    }  
}  
  
// Muestra la cola genérica.  
class GenColaDemo {  
    static void Main() {
```

(continúa)

```
// Crea una cola para chars y una cola para doubles.  
ColaSimple<char> charQ = new ColaSimple<char>(10);  
ColaSimple<double> doubleQ = new ColaSimple<double>(5);  
  
char ch;  
double d;  
int i;  
  
try {  
    // Usa la cola char.  
    for(i=0; i < 10; i++) {  
        Console.WriteLine("Almacena: " + (char)('A' + i));  
        charQ.Put((char)('A' + i));  
    }  
    Console.WriteLine();  
  
    for(i=0; i < 10; i++) {  
        Console.Write("Obtiene el siguiente carácter: ");  
        ch = charQ.Get();  
        Console.WriteLine(ch);  
    }  
}  
catch (ColaLLenaException exc) {  
    Console.WriteLine(exc);  
}  
Console.WriteLine();  
  
try {  
    // Usa una cola para double.  
    for(i=1; i <= 5; i++) {  
        Console.WriteLine("Almacena: " + i * 3.1416);  
        doubleQ.Put(i * 3.1416);  
    }  
  
    Console.WriteLine();  
  
    for(i=0; i < 5; i++) {  
        Console.Write("Obtiene el siguiente double: ");  
        d = doubleQ.Get();  
        Console.WriteLine(d);  
    }  
}  
catch (ColaVaciaException exc) {  
    Console.WriteLine(exc);  
}  
}
```

Los datos generados por el programa son los siguientes:

```
Almacena: A
Almacena: B
Almacena: C
Almacena: D
Almacena: E
Almacena: F
Almacena: G
Almacena: H
Almacena: I
Almacena: J
```

```
Obtiene el siguiente carácter: A
Obtiene el siguiente carácter: B
Obtiene el siguiente carácter: C
Obtiene el siguiente carácter: D
Obtiene el siguiente carácter: E
Obtiene el siguiente carácter: F
Obtiene el siguiente carácter: G
Obtiene el siguiente carácter: H
Obtiene el siguiente carácter: I
Obtiene el siguiente caracter: J
```

```
Almacena: 3.1416
Almacena: 6.2832
Almacena: 9.4248
Almacena: 12.5664
Almacena: 15.708
```

```
Obtiene el siguiente double: 3.1416
Obtiene el siguiente double: 6.2832
Obtiene el siguiente double: 9.4248
Obtiene el siguiente double: 12.5664
Obtiene el siguiente double: 15.708
```

---

## ✓ Autoexamen Capítulo 13

- 1.** Los genéricos mejoran la eficiencia de las clases que funcionan con varios tipos de datos manteniendo la seguridad de tipos, ¿cierto o falso?
- 2.** ¿Qué es un parámetro de tipo?
- 3.** ¿Qué es un argumento de tipo?

4. Como se explicó, el código generalizado fue posible en C# antes de añadir los genéricos a través del uso de referencias **object**. Un problema con este enfoque es el tedio y la posibilidad de errores asociados con realizar la transformación de tipo requerida. ¿Qué otro problema presentaba?
5. ¿Qué hace una limitación de clase? ¿Qué hace una limitación de tipo valor?
6. ¿Las interfaces pueden ser genéricas?
7. ¿Cómo creas un valor por defecto para un tipo?
8. Cuando se crea un método genérico, ¿dónde se debe especificar la lista de parámetros?
9. ¿Los métodos genéricos pueden utilizar limitaciones de tipo?
10. Como se mencionó, aunque no se pueden crear propiedades genéricas, una propiedad puede utilizar un parámetro de tipo para su alcance cerrado. Vuelve a codificar la interfaz **IDosDCoord** y la clase **XYCoord** mostradas en este capítulo de manera que conviertan métodos como **GetX()** y **GetY()** en propiedades.

# Capítulo 14

## Introducción a LINQ

## Habilidades y conceptos clave

- Fundamentos de LINQ
  - Correspondencia de tipos en consultas
  - Filtrar valores con **where**
  - Ordenar resultados con **orderby**
  - Seleccionar valores con **select**
  - Agrupar resultados con **group**
  - Usar **into** para crear una continuación
  - Unir dos secuencias con **join**
  - Tipos anónimos e inicializadores de objetos
  - Crear un grupo unido
  - Usar **let** para crear una variable en una consulta
  - Los métodos **query**
  - Ejecución inmediata *vs.* diferida en un query
  - Expresiones lambda
  - Métodos de extensión
- 

**S**i la adición de genéricos en la versión 2.0 tuvo un profundo efecto en C# (lo cual sucedió), la posterior adición de LINQ en la versión 3.0 ¡no es menos que un terremoto! Sin lugar a dudas, LINQ es la característica individual más importante en C# 3.0. Añade un elemento sintáctico completamente nuevo, varias palabras clave nuevas y una poderosa capacidad sin precedente. La inclusión de LINQ ha incrementado significativamente los alcances del lenguaje, expandiendo el rango de tareas en las cuales puede aplicarse C#. En términos sencillos, con la introducción de LINQ, C# ha establecido un nuevo estándar, que afectará el desarrollo de los lenguajes en el futuro cercano. Así de importante es LINQ.

### **NOTA**

Como en el caso de los genéricos descritos en el capítulo anterior, el tema de LINQ es muy largo; involucra muchas características, opciones y alternativas. No es posible cubrir todos sus aspectos en esta guía para principiantes. (De hecho, una descripción completa de LINQ ¡requeriría un libro completo dedicado a ello!) Este capítulo explica su teoría, los conceptos clave y la sintaxis básica. También muestra varios ejemplos que ilustran el uso de LINQ. Después de

terminar este capítulo, estarás en condiciones de utilizar LINQ en tus programas. Sin embargo, es un subsistema que con seguridad querrás explorar con mayor detalle.

## ¿Qué es LINQ?

LINQ significa lenguaje integrado de consulta (query).<sup>1</sup> Está constituido por un conjunto de características que permiten recuperar información de una fuente de datos. Como probablemente sepas, la recuperación de datos constituye una parte importante de muchos programas. Por ejemplo, un programa puede obtener información de una lista de clientes, ver la información de un catálogo de productos o accesar al registro de un empleado. En muchos casos, esos datos están almacenados en una base de datos que está separada de la aplicación. Por ejemplo, un catálogo de productos puede estar contenido en una base de datos relacional. En el pasado, interactuar con una de ellas implicaría generar queries utilizando SQL (Lenguaje Estructurado de Consultas). Otro tipo de datos, como XML, requieren sus propias herramientas. Por tanto, antes de C# 3.0, el soporte para esos queries no estaba integrado en C#. LINQ cambia todo esto.

LINQ añade a C# la capacidad de generar queries para cualquier fuente de datos compatible con LINQ. Más aún, la sintaxis utilizada para la consulta es la misma, sin importar la fuente de datos que se utilice. Esto significa que la sintaxis que se utiliza para consultar datos en una base de datos relacional es la misma que la utilizada para realizar una consulta en datos almacenados en un arreglo, por ejemplo. No es necesario utilizar SQL o cualquier otro mecanismo ajeno a C#. La capacidad de consultas está completamente integrada en el lenguaje C#.

Además de utilizar LINQ con SQL, también puede usarse con archivos XML y bases de datos ADO.NET. Tal vez de la misma importancia es que también puede emplearse con arreglos C# y colecciones (descritas en el capítulo 15). Por tanto, LINQ te ofrece un medio uniforme para accesar datos. Éste es un concepto poderoso e innovador. No sólo modifica la manera de accesar datos, también ofrece una nueva manera de razonar sobre cómo abordar problemas antiguos. En el futuro, muchos tipos de soluciones informáticas serán elaboradas en términos de LINQ. Sus efectos no se limitarán sólo a bases de datos.

LINQ es soportado por un conjunto de características interrelacionadas, incluyendo la sintaxis de consultas añadidas al lenguaje C#, expresiones lambda, tipos anónimos y métodos de extensión. Todas ellas son examinadas en este capítulo.

## Fundamentos de LINQ

En el núcleo de LINQ está la consulta. Éste especifica qué datos se recuperarán de una fuente de datos. Por ejemplo, una consulta en una lista de correo personalizada puede solicitar la dirección de todos los clientes que residen en una ciudad específica, como Chicago o Tokio. Un query aplicado a una base de datos de inventario puede solicitar una lista de las mercancías que se han terminado. Si se aplica a una bitácora de uso de Internet puede pedir una lista de las estaciones Web con la mayor cantidad de visitas. Aunque estas consultas difieren en sus detalles, todos ellos pueden expresarse utilizando los mismos elementos sintácticos de LINQ.

---

<sup>1</sup> *Query* es una *consulta* a una base de datos para recuperar información, añadirla o modificarla. Se respeta el término en inglés porque es el más utilizado en el argot cotidiano de desarrolladores; durante este capítulo se alternarán los términos *consulta* y *query*.

Después de crear una consulta, se puede ejecutar. Una manera de realizar esta tarea es utilizando el query en un loop **foreach**. Su ejecución da como resultado la obtención de los datos solicitados. Así, su uso involucra dos pasos clave. Primero, se crea el formato del query. Segundo, se ejecuta. Por ello, la consulta define *qué* recuperar de la fuente de datos. La ejecución del query es el mecanismo por el cual, de hecho, *se obtienen los resultados*.

Para que una fuente de datos pueda ser utilizada por LINQ, debe implementar la interfaz **IEnumerable**. Existen dos formatos de esta interfaz: uno genérico y otro no genérico. En general, la tarea es más fácil si la fuente de datos implementa la versión genérica, **IEnumerable<T>**, donde **T** especifica el tipo de dato que se va a enumerar. El resto del capítulo da por hecho que la fuente de datos implementa **IEnumerable<T>**. Esta interfaz está declarada en **System.Collections.Generic**. Una clase que implementa **IEnumerable<T>** soporta enumeraciones, lo que significa que su contenido puede ser obtenido uno a la vez en secuencia. Todos los arreglos C# soportan **IEnumerable<T>**. De esta manera, podemos utilizar arreglos para mostrar los conceptos centrales de LINQ. Debes entender, sin embargo, que LINQ no se limita a los arreglos.

## Un query sencillo

Antes de continuar con la teoría, trabajemos con un ejemplo sencillo de LINQ. El siguiente programa utiliza un query para obtener los valores positivos contenidos en un arreglo de enteros:<sup>2</sup>

```
// Crea un query LINQ sencillo.
using System;
using System.Linq;

class SimpQuery {
    static void Main() {

        int[] nums = { 1, -2, 3, 0, -4, 5 };

        // Crea un query que obtiene sólo los números positivos.
        var posNums = from n in nums
                      where n > 0           ← Crea un query.
                      select n;

        Console.WriteLine("Los valores positivos en nums:");

        // Ejecuta el query y muestra los resultados.
        foreach(int i in posNums) Console.WriteLine(i); ← Ejecuta el query.
    }
}
```

---

<sup>2</sup> Si estás utilizando la versión 3.5 o superior de Microsoft Visual C# 2008 Express, para que funcionen las características LINQ es **indispensable** que agregues la referencia **System.Core** a C#. Para hacerlo, haz clic con el botón derecho sobre el nombre del programa en la ventana “Explorador de soluciones”, selecciona “Agregar referencia”. Aparecerá una ventana emergente. En la pestaña “.NET” haz clic sobre la referencia **System.Core**. Finalmente haz clic sobre el botón “Aceptar”. En algunos casos es posible que también necesites agregar la referencia **System.Data.Linq**; sigue el procedimiento anterior para activarla.

El programa genera los siguientes resultados:

```
Los valores positivos en nums:  
1  
3  
5
```

Como puedes ver, se muestran únicamente los valores positivos del arreglo **nums**. Aunque muy sencillo, este programa muestra las características clave de LINQ. Examinémoslo de cerca.

Lo primero que debes observar es la directiva **using** del programa:

```
using System.Linq;
```

Para utilizar las características LINQ, debes incluir la nomenclatura **System.Linq**.

A continuación, se declara un arreglo de **int** llamado **nums**. Todos los arreglos en C# son implícitamente convertibles a **IEnumerable<T>**. Esto hace que cualquier arreglo de C# sea utilizable como fuente de datos para LINQ.

Paso siguiente, se declara un query que recupera los elementos en **nums** que son positivos. Como se muestra aquí:

```
var posNums = from n in nums  
              where n > 0  
              select n;
```

La variable **posNums** es llamada la *variable del query*. Hace referencia al conjunto de reglas definidas por la consulta. Observa que utiliza **var** para declarar implícitamente **posNums**. Como sabes, esto la convierte en una variable de tipo implícito. En los queries, muchas veces es conveniente utilizar variables de tipo implícito, aunque también puedes declarar explícitamente el tipo, el cual debe ser una forma de **IEnumerable<T>**. Luego, la variable **posNums** es asignada a la expresión del query.

Todas las consultas comienzan con **from**. Esta cláusula especifica dos elementos. El primero es la *variable de rango*, que recibirá los elementos obtenidos de la fuente de datos. En este caso, la variable de rango es **n**. El segundo elemento es la fuente de datos, la cual es, en este caso, el arreglo **nums**. El tipo de la variable de rango es inferido a partir de la fuente de datos. En este caso, el tipo de **n** es **int**. Generalizando, ésta es la sintaxis de la cláusula **from**:

```
from variable-rango in fuente-datos
```

La siguiente cláusula en el query es **where**. Especifica una condición que un elemento en la fuente de datos debe cumplir para que sea seleccionado por la consulta. Su formato general se muestra a continuación:

```
where expresión-boleana
```

La *expresión-boleana* debe producir un resultado **bool**. (Esta expresión también es conocida como *predicado*.) Un query puede tener más de una cláusula **where**. En el programa se utiliza la siguiente cláusula **where**:

```
where n > 0
```

Será verdadera sólo para aquellos elementos cuyos valores sean mayores que cero. Esta expresión será evaluada por cada **n** en **nums** cuando se ejecute la consulta. Sólo aquellos valores que satis-

fagan esta condición serán recuperados. En otras palabras, una cláusula **where** actúa como filtro sobre una fuente de datos, permitiendo sólo el paso de ciertos elementos.

Todos los queries terminan con una cláusula **select** o una cláusula **group**. En este ejemplo se utiliza la primera. Especifica exactamente lo que se obtiene con el query. Para consultas sencillas, como la del ejemplo, se selecciona un rango de valores. Por ello, regresa aquellos enteros del arreglo **nums** que satisfacen lo establecido por la cláusula **where**. En situaciones más sofisticadas, es posible afinar más los datos seleccionados. Por ejemplo, cuando se consulta una lista de correo electrónico, es posible que quieras recuperar sólo el apellido de cada contacto, en lugar de toda la dirección. Observa que la cláusula **select** finaliza con punto y coma. Dado que **select** finaliza la consulta, da por terminada toda la declaración y requiere ese punto y coma. Observa también que las demás cláusulas en la consulta no finalizan con ese signo de puntuación.

En este punto, se ha creado una variable de consulta llamada **posNums**, pero no se ha obtenido ningún resultado. Es importante comprender que la consulta sólo define un conjunto de reglas. Sólo cuando se ejecuta el mismo se obtienen resultados. Más aún, el mismo query puede ser ejecutado dos o más veces, con la posibilidad de obtener datos diferentes si la fuente de datos subyacente se modifica entre las ejecuciones. Por lo mismo, simplemente declarar el query **posNums** no significa que contiene el resultado de la consulta.

Para ejecutar el query, el programa utiliza el loop **foreach** que se muestra aquí:

```
foreach (int i in posNums) Console.WriteLine(i);
```

Observa que **posNums** se especifica como la colección de datos sobre la cual el loop realizará las reiteraciones. Cuando se ejecuta **foreach**, se ejecutan las reglas definidas por el query específicamente para **posNums**. Con cada reiteración del loop, se obtiene el siguiente elemento regresado por la consulta. El proceso finaliza cuando no hay más elementos por recuperar. En este caso, el tipo de la variable de reiteración **i** es especificado explícitamente como **int** porque ése es el tipo de elementos recuperados por el query. En esta situación, es correcto especificar explícitamente el tipo de variable de reiteración, porque es fácil saber el tipo de la variable seleccionada por la consulta. Sin embargo, en situaciones más complejas, será más fácil (y en algunos casos necesario) especificar implícitamente el tipo de la variable de reiteración utilizando **var**.

## Un query puede ser ejecutado más de una vez

Como un query define un conjunto de reglas que se utilizan para recuperar datos, pero en sí no produce resultados, la misma consulta puede ejecutarse múltiples veces. Si la fuente de datos se modifica entre ejecuciones, entonces los resultados pueden ser diferentes. Por lo mismo, una vez que defines el query, su ejecución siempre producirá el resultado más reciente. He aquí un ejemplo. En la siguiente versión del programa anterior, el contenido del arreglo **nums** se modifica entre dos ejecuciones de **posNums**:

```
// Crea un query simple.  
using System;  
using System.Linq;  
using System.Collections.Generic;  
  
class SimpQuery {  
    static void Main() {
```

```

int[] nums = { 1, -2, 3, 0, -4, 5 };
// Crea un query que obtiene sólo los números positivos.
var posNums = from n in nums
              where n > 0
              select n;

Console.WriteLine("Los valores positivos en nums:");
// Ejecuta el query y muestra los resultados.
foreach(int i in posNums) Console.WriteLine(i); ←

// Cambia nums.
Console.WriteLine("\nEstablece nums[1] a 99.");
nums[1] = 99;
Console.WriteLine("Los valores positivos en nums:");
// Ejecuta el query por segunda ocasión.
foreach(int i in posNums) Console.WriteLine(i); ←
}
}

```

Cada **foreach** produce un resultado diferente.

El programa genera los siguientes datos:

```

Los valores positivos en nums:
1
3
5
Establece nums[1] a 99.
Los valores positivos en nums:
1
99
3
5

```

Como lo confirman los datos de salida, después de que el valor de **nums[1]** cambió de **-2** a **99**, el resultado de volver a ejecutar la consulta refleja estos cambios. Éste es un punto clave que debe enfatizarse. Cada ejecución produce sus propios resultados, que son obtenidos al enumerar el contenido actual de la fuente de datos. Por lo mismo, si la fuente de datos se modifica, también puede hacerlo el resultado de la ejecución del query. Los beneficios de este enfoque son muy significativos. Por ejemplo, si estás recuperando una lista de órdenes pendientes para una tienda en línea, querrás que cada ejecución de tu consulta produzca todas las órdenes, incluyendo aquellas que acaban de entrar.

## Cómo se relacionan los tipos de datos en un query

Como lo han mostrado los ejemplos anteriores, una consulta implica el uso de variables cuyos tipos se relacionan entre sí. Éstas son la variable de query, la variable de rango y la fuente de datos. Dada la correspondencia entre estos tipos y la importancia de esta relación, además de que puede ser un poco confuso al principio, vale la pena mirarlos más de cerca.

El tipo de la variable de rango debe coincidir con el tipo de elementos almacenados en la fuente de datos. Así, el tipo de la variable de rango es dependiente del tipo de la fuente de datos. En muchos casos, C# puede inferir el tipo de la variable de rango. Mientras que la fuente de datos implemente **IEnumerable<T>**, se puede realizar la inferencia de datos, porque **T** describe el tipo de los elementos en la fuente. (Como ya se mencionó, todos los arreglos implementan **IEnumerable<T>**, lo mismo que muchas otras fuentes de datos.) No obstante, si la fuente de datos implementa la versión no genérica de **IEnumerable**, entonces necesitarás especificar explícitamente el tipo de la variable de rango. Esto se hace especificando su tipo en la cláusula **from**. Por ejemplo, retomando los ejemplos anteriores, la siguiente línea muestra cómo declarar explícitamente **n** para que sea **int**:

```
var posNums = from int n in nums  
    / / ...
```

Por supuesto, la especificación explícita de tipo no es necesaria aquí, porque todos los arreglos son implícitamente transformables a **IEnumerable<T>**, lo cual permite que el tipo de la variable de rango sea inferido.

El tipo de objeto regresado por una consulta es una instancia de **IEnumerable<T>**, donde **T** es el tipo de los elementos. De esta manera, el tipo de la variable de query debe ser una instancia de **IEnumerable<T>**. El valor de **T** es determinado por el tipo de valor especificado por la cláusula **select**. En el ejemplo anterior, **T** es un **int** porque **n** es un **int**. (Como ya se explicó, **n** es un **int** porque **int** es el tipo de los elementos almacenados en **nums**.) Por ello, el query pudo ser escrito de la siguiente manera, con la especificación explícita del tipo como **IEnumerable<int>**:

```
IEnumerable<int> posNums = from n in nums  
    where n > 0  
    select n;
```

El punto clave es que el tipo del elemento seleccionado por **select** debe coincidir con el tipo de argumento transmitido a **IEnumerable<T>** utilizado para declarar la variable de query. Muy a menudo, éstas variables utilizan **var** en vez de la especificación explícita de tipo porque esto permite que el compilador infiera el tipo apropiado de la cláusula **select**. Como verás, este enfoque es particularmente útil cuando **select** regresa algo diferente a un elemento dentro de una fuente de datos.

Cuando se ejecuta un query por el loop **foreach**, el tipo de la variable de reiteración debe ser el mismo que el tipo de la variable de rango. En los ejemplos anteriores, este tipo fue especificado explícitamente como **int**, pero puedes dejar que el compilador infiera el tipo especificando esta variable como **var**. Como verás, existen algunos casos en los cuales debe utilizarse **var** porque el nombre del tipo de dato es desconocido.

## El formato general de un query

Todos los queries comparten un formato general, que se basa en un conjunto de palabras clave contextuales, mostradas a continuación:

|           |            |         |        |
|-----------|------------|---------|--------|
| ascending | descending | equals  | from   |
| group     | in         | into    | join   |
| let       | on         | orderby | select |
| where     |            |         |        |

De ellas, las siguientes son utilizadas para dar inicio a una cláusula query:

|         |        |       |     |
|---------|--------|-------|-----|
| from    | group  | join  | let |
| orderby | select | where |     |

Como ya se mencionó, un query debe comenzar con la palabra clave **from** y finalizar con cualquiera de las cláusulas **select** o **group**. La cláusula **select** determina qué tipo de valores son enumerados por el query. La cláusula **group** regresa los datos por grupos, cada grupo con la posibilidad de ser enumerado individualmente. Como lo han mostrado los ejemplos anteriores, la cláusula **where** especifica el criterio que un elemento debe cumplir para que sea parte de los datos seleccionados y regresados. Las demás cláusulas te ayudan a afinar la consulta. Las siguientes secciones examinan cada una de sus cláusulas.

## Filtrar valores con where

Como ya se explicó, la cláusula **where** es utilizada para filtrar los datos regresados por la consulta. Los ejemplos anteriores han mostrado sólo su formato más sencillo, en el cual se utiliza sólo una condición. Un punto clave a entender es que puedes usar **where** para filtrar datos con base en más de una condición. Una manera de hacerlo es utilizando múltiples cláusulas **where**. Por ejemplo, analiza el siguiente programa que muestra sólo aquellos valores en el arreglo que son tanto positivos como menores que 10.

```
// Utiliza múltiples cláusulas where.
using System;
using System.Linq;

class DosWhere {
    static void Main() {
        int[] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };

        // Crea un query que obtiene valores positivos menores a 10.
        var posNums = from n in nums
                      where n > 0
                      where n < 10
                      select n;
        ←———— Utiliza dos cláusulas where.

        Console.WriteLine("Los valores positivos menores a 10:");
        // Ejecuta el query y muestra los resultados.
        foreach(int i in posNums) Console.WriteLine(i);
    }
}
```

Los datos generados por el programa son:

```
Los valores positivos menores a 10:
1
3
6
9
```

Como puedes ver, sólo se recuperan los valores positivos menores que 10.

Aunque es válido utilizar dos cláusulas **where** como se acaba de mostrar, se puede obtener el mismo resultado de una manera más compacta utilizando un **where** en el cual ambas validaciones estén combinadas en una misma expresión. Aquí se presenta el query modificado para utilizar esta perspectiva:

```
var posNums = from n in nums
               where n > 0 && n < 10
               select n;
```

En general, una condición **where** puede utilizar cualquier expresión C# válida que evalúe un resultado booleano. Por ejemplo, el siguiente programa define un arreglo **string**. Varias de las cadenas de caracteres son direcciones Internet. El query **netDirec** recupera sólo aquellas cadenas que tienen más de cuatro caracteres y que terminan con “.net”. De esa manera, encuentra aquellas cadenas que contienen una dirección Internet que utiliza el nombre de dominio “.net”.

```
// Muestra el funcionamiento de otra cláusula where.
using System;
using System.Linq;

class WhereDemo2 {

    static void Main() {

        string[] strs = { ".com", ".net", "hsNombreA.com", "hsNombreB.net",
                          "test", ".network", "hsNombreC.net", "hsNombreD.com" };

        // Crea un query que obtiene direcciones Internet
        // que finalicen con .net.
        var netDirec = from direc in strs
                       where direc.Length > 4 && direc.EndsWith(".net")
                       select direc;

        // Ejecuta el query y muestra los resultados.
        foreach(var str in netDirec) Console.WriteLine(str);
    }
}
```

↑  
Una expresión **where**  
más compleja.

Los datos generados por el programa son:

```
hsNombreB.net
hsNombreC.net
```

Observa que el programa hace uso de otro método **string** llamado **EndsWith( )** (*FinalizaCon*). Regresa un valor verdadero si la cadena invocante finaliza con la secuencia de caracteres especificada como argumento.

## Ordenar resultados con orderby

Con frecuencia querrás que los resultados de la consulta aparezcan ordenados. Por ejemplo, quizás quieras obtener una lista de cuentas vencidas, para mantener el balance restante, de ma-

yor a menor. O bien, quizás quieras obtener una lista de clientes en orden alfabético por nombre. Cualquiera que sea el propósito, LINQ te ofrece un medio fácil para producir resultados ordenados: la cláusula **orderby**.

El formato general de **orderby** se muestra a continuación:

**orderby** *ordenar-en* *cómo*

El elemento sobre el cual se aplicará el orden es especificado por *ordenar-en*. Esto puede ser tan amplio como todos los elementos almacenados en la fuente de datos o tan restringido como la porción de un solo campo dentro de los elementos. El valor de *cómo* determina si el orden es ascendente o descendente, y debe ser una de las dos opciones: **ascending** o **descending**, respectivamente. El orden por defecto es ascendente, así que por lo general no especificarás la opción **ascending**.

He aquí un ejemplo que utiliza **orderby** para recuperar los valores en un arreglo **int** en orden ascendente:

```
// Muestra orderby en acción.
using System;
using System.Linq;

class OrderbyDemo {

    static void Main() {

        int[] nums = { 10, -19, 4, 7, 2, -5, 0 };

        // Crea un query que obtiene los valores en orden.
        var sNums = from n in nums
                    orderby n ← Ordena el resultado.
                    select n;

        Console.WriteLine("Valores en orden: ");

        // Ejecuta el query y muestra los resultados.
        foreach(int i in sNums) Console.Write(i + " ");
        Console.WriteLine();
    }
}
```

Los datos generados por el programa son:

```
Valores en orden ascendente: -19 -5 0 2 4 7 10
```

Para cambiar el orden a descendente, simplemente especifica la opción **descending**, como se muestra a continuación:

```
var sNums = from n in nums
            orderby n descending
            select n;
```

Si intentas esta opción, verás que el orden de los valores se invierte.

## Una mirada cercana a select

La cláusula **select** determina qué tipo de elementos se obtienen con la consulta. Su formato general se muestra a continuación:

`select expresión`

Hasta ahora, hemos utilizado **select** para recuperar la variable de rango. Así, *expresión* ha nombrado simplemente la variable de rango. Sin embargo, **select** no se limita a esa simple acción. Puede regresar una porción específica de la variable de rango, el resultado de aplicar alguna operación o transformación a la variable de rango, e incluso un nuevo tipo de objeto que se construya a partir de las piezas de la información obtenida de la variable de rango. A esto se le llama *proyectar*.

Para comenzar a examinar las otras capacidades de **select**, analiza el siguiente programa. Muestra las raíces cuadradas de los valores positivos contenidos en un arreglo de valores **double**.

```
// Utiliza select para recuperar las raíces cuadradas de todos los
valores positivos
// en un arreglo de doubles.
using System;
using System.Linq;

class SelectDemo {

    static void Main() {

        double[] nums = { -10.0, 16.4, 12.125, 100.85, -2.2, 25.25, -3.5 };

        // Crea un arreglo que recupera las raíces cuadradas de todos los
        // valores positivos en nums.
        var raizCuadrada = from n in nums
                            where n > 0
                            select Math.Sqrt(n); ← Regresa una secuencia que contiene
   las raíces cuadradas de n.

        Console.WriteLine("Las raíces cuadradas de los valores positivos" +
                          " redondeadas a dos lugares decimales:");

        // Ejecuta el query y muestra los resultados.
        foreach(double r in raizCuadrada) Console.WriteLine(" {0:#.##}", r);
    }
}
```

Los datos generados por el programa son:

Las raíces cuadradas de los valores positivos redondeadas a dos lugares decimales:  
4.05  
3.48  
10.04  
5.02

En el programa, presta especial atención a la cláusula **select**:

```
select Math.Sqrt(n);
```

Regresa la raíz cuadrada de la variable de rango. Lo hace obteniendo el resultado de transmitir la variable de rango a **Math.Sqrt()**, que regresa la raíz cuadrada de sus argumentos. Esto significa que la secuencia obtenida cuando la consulta se ejecuta contendrá la raíz cuadrada de los valores positivos en **nums**. Si generalizas este concepto, se manifiesta el poder de **select**. Puedes utilizar **select** para generar cualquier tipo de secuencia que necesites, con base en los valores obtenidos de la fuente de datos.

A continuación presentamos un programa que muestra otra manera de utilizar **select**. Crea una clase llamada **EmailDirec** que contiene dos propiedades. La primera almacena el nombre de la persona. La segunda contiene la dirección de correo electrónico. Luego, el programa crea un arreglo que contiene varias entidades **EmailDirec**. El programa utiliza una consulta para obtener una lista de sólo las direcciones de correo electrónico, sin los nombres.

```
// Regresa una porción de la variable de rango.
using System;
using System.Linq;

class EmailDirec {
    public string Nombre { get; set; }
    public string Dirección { get; set; }

    public EmailDirec(string n, string a) {
        Nombre = n;
        Dirección = a;
    }
}

class SelectDemo2 {
    static void Main() {

        EmailDirec[] direc = {
            new EmailDirec("Herb", "Herb@HerbSchildt.com"),
            new EmailDirec("Tom", "Tom@HerbSchildt.com"),
            new EmailDirec("Sara", "Sara@HerbSchildt.com")
        };

        // Crea un query que selecciona la dirección de correo electrónico.
        var eDirec = from entry in direc
                     select entry.Dirección; ← Utiliza sólo la porción Dirección
   de los elementos en direc.

        Console.WriteLine("Las direcciones e-mail son");

        // Ejecuta el query y muestra los resultados.
        foreach(string s in eDirec) Console.WriteLine(" " + s);
    }
}
```

Los datos generados por el programa son:

```
Las direcciones e-mail son
Herb@HerbSchildt.com
Tom@HerbSchildt.com
Sara@HerbSchildt.com
```

Pon especial atención a la cláusula **select**:

```
select entry.Direccion;
```

En vez de regresar la variable de rango entera, regresa solamente una porción de **Dirección**. Este hecho lo corroboran los datos de salida. Esto significa que la consulta regresa una secuencia de cadenas de caracteres, no una secuencia de objetos **EmailDirec**. Por ello el loop **foreach** especifica **s** como **string**. Como ya se explicó, el tipo de secuencias regresado por el query lo determina el tipo del valor regresado por la cláusula **select**.

Una de las características más poderosas de **select** es su capacidad para regresar una secuencia que contenga elementos creados durante la ejecución de la consulta. Por ejemplo, analiza el siguiente programa. Define una clase llamada **ContactInfo**, que almacena un nombre, dirección de correo electrónico y número telefónico. También define la clase **EmailDirec** utilizada en el ejemplo anterior. Dentro de **Main()**, se crea un arreglo de **ContactInfo**. Luego, se declara un query en el cual la fuente de datos es un arreglo de **ContactInfo**, pero la secuencia regresada contiene objetos **EmailDirec**. Así, el tipo de la secuencia regresado por **select** no es **ContactInfo**, sino **EmailDirec**, y estos objetos son creados durante la ejecución del query.

```
// Utiliza un query para obtener una secuencia de EmailDirec
// de una lista de ContactInfo.
using System;
using System.Linq;

class ContactInfo {
    public string Nombre { get; set; }
    public string Email { get; set; }
    public string Tel { get; set; }

    public ContactInfo(string n, string a, string p) {
        Nombre = n;
        Email = a;
        Tel = p;
    }
}

class EmailDirección {
    public string Nombre { get; set; }
    public string Dirección { get; set; }

    public EmailDirección(string n, string a) {
        Nombre = n;
```

```

        Dirección = a;
    }
}

class SelectDemo3 {
    static void Main() {

        ContactInfo[] contactos = {
            new ContactInfo("Herb", "Herb@HerbSchildt.com", "555-1010"),
            new ContactInfo("Tom", "Tom@HerbSchildt.com", "555-1101"),
            new ContactInfo("Sara", "Sara@HerbSchildt.com", "555-0110")
        };

        // Crea un query que forma una lista de objetos EmailDirección.
        var emailList = from entry in contactos
                        select new EmailDirección(entry.Nombre, entry.Email);

        Console.WriteLine("La lista e-mail");

        // Ejecuta el query y muestra los resultados.
        foreach(EmailDirección e in emailList)
            Console.WriteLine(" {0}: {1}", e.Nombre, e.Dirección );
    }
}

```

A partir del arreglo **ContactInfo**, produce una lista de objetos **EmailDirección**.

Los datos generados por el programa son:

```

La lista e-mail
Herb: Herb@HerbSchildt.com
Tom: Tom@HerbSchildt.com
Sara: Sara@HerbSchildt.com

```

El punto clave de este ejemplo es que el tipo de secuencias generado por el query puede consistir en objetos creados por él mismo.

## Agrupar resultados con group

Una de las características más poderosas del query la proporciona la cláusula **group**, porque te permite crear resultados que están agrupados por claves. Utilizando la secuencia obtenida a partir de un grupo, puedes accesar fácilmente todos los datos asociados con esa clave. Esto hace que **group** sea una manera fácil y efectiva de recuperar datos que están organizados en secuencias de elementos relacionados. La cláusula **group** es una de las dos cláusulas que finalizan la consulta. (La otra es **select**.)

La cláusula **group** tiene el siguiente formato general:

*group variable-rango by clave*

Regresa datos agrupados en secuencias, cada una de las cuales tiene en común la característica especificada por *clave*.

El resultado de **group** es una secuencia que contiene elementos de tipo **IGrouping< TKey, TElement >**, que está declarado en la nomenclatura **System.Linq**. Define una colección de objetos que comparten una clave en común. El tipo de la variable query en una consulta que regresa un grupo es **IEnumerable< IGrouping< TKey, TElement > >**. **IGrouping** define una propiedad sólo-lectura llamada **Key**, que regresa la clave asociada con cada secuencia.

A continuación presentamos un ejemplo que ilustra el uso de **group**. Declara un arreglo que contiene una lista de sitios Web. Luego crea un query que agrupa la lista por el nombre superior del dominio, como **.org** o **.com**.

```
// Muestra la cláusula group en acción.
using System;
using System.Linq;

class GroupDemo {

    static void Main() {

        string[] sitiosweb = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                               "hsNameD.com", "hsNameE.org", "hsNameF.org",
                               "hsNameG.tv", "hsNameH.net", "hsNameI.tv" };

        // Crea un query que agrupa los sitios Web por nombre superior de dominio.
        var DirWeb = from dir in sitiosweb
                     where dir.LastIndexOf(".") != -1
                     group dir by dir.Substring(dir.LastIndexOf("."));

        // Ejecuta el query y muestra los resultados.
        foreach(var sitios in DirWeb) {
            Console.WriteLine("Sitios Web agrupados por " + sitios.Key);
            foreach(var sitio in sitios)
                Console.WriteLine(" " + sitio);
            Console.WriteLine();
        }
    }
}
```

↑  
Agrupa los resultados por  
nombre de dominio.

Los resultados generados por el programa son:

```
Sitios Web agrupados por .com
hsNameA.com
hsNameD.com
```

```
Sitios Web agrupados por .net
hsNameB.net
hsNameC.net
hsNameH.net
```

```
Sitios Web agrupados por.org
```

```
hsNameE.org
hsNameF.org
```

```
Sitios Web agrupados por .tv
hsNameG.tv
hsNameI.tv
```

Como lo muestran los datos de salida, los datos se agrupan con base en el nombre superior de dominio de las estaciones Web. Observa cómo se realiza esto utilizando la cláusula **group**:

```
var DirWeb = from dir in sitiosweb
             where dir.LastIndexOf(".") != -1
             group dir by dir.Substring(dir.LastIndexOf("."));
```

La clave se obtiene utilizando los métodos **LastIndexOf()** y **Substring()** definidos por **string**. (Éstos son explicados en el capítulo 5. La versión de **Substring()** utilizada aquí regresa la subcadena que comienza en el índice especificado y corre hasta el final de la cadena invocante.) El índice del último punto en el nombre de la estación Web se localiza con el uso del método **LastIndexOf()**. Utilizando este índice, el método **Substring()** obtiene lo que resta de la cadena, que es la parte del nombre de la estación Web que contiene el nombre superior de dominio. Otro punto: observa el uso de la cláusula **where** para filtrar cualquier cadena que no contiene un punto. El método **LastIndexOf()** regresa -1 si la cadena especificada no está contenida en la cadena invocante.

Como la cadena que se obtiene cuando se ejecuta en **DirWeb** es una lista de grupos, necesitas utilizar dos loop **foreach** para accesar los miembros de cada grupo. El loop externo obtiene cada grupo. El loop interno enumera los miembros dentro del grupo. La variable de reiteración del loop **foreach** externo debe ser una instancia **IGrouping** compatible con los tipos de la clave y los elementos. En el ejemplo, tanto la clave como los elementos son de tipo **string**. Por lo mismo, el tipo de la variable de reiteración **sitios** del loop externo es **IGrouping<string, string>**. El tipo de la variable de reiteración del loop interno es **string**. Por cuestiones de brevedad, el ejemplo declara implícitamente estas variables, pero pudieron ser declaradas explícitamente como se muestra a continuación:

```
foreach(IGrouping<string, string> sitios in DirWeb) {
    Console.WriteLine("Sitios Web agrupados por" + sitios.Key);
    foreach(string sitio in sitios)
        Console.WriteLine(" " + sitio);
    Console.WriteLine();
}
```

## Utilizar **into** para crear una continuidad

Cuando utilizas **select** o **group**, en ocasiones querrás generar resultados temporales que sean utilizados por una parte subsecuente de la consulta para producir un resultado final. Esta acción recibe el nombre de *continuidad de query* (o simplemente *continuidad* para abreviar), y se realiza utilizando la palabra clave **into** con las cláusulas **select** o **group**. Tiene el siguiente formato general:

*into nombre cuerpo-query*

donde *nombre* es el nombre de la variable de rango que reitera sobre el resultado temporal y es utilizada por el query de continuidad, especificado por *cuerpo-query*. Por esta razón **into** es llamado continuidad de query cuando se utiliza con **select** o **group**: le da continuidad a la consulta. En esencia, un query de continuidad representa el concepto de construir un nuevo query que consulta el resultado anterior.

### NOTA

También existe un formato de **into** que puede ser utilizado con **join**, que crea un *grupo de unión*. Éste se describe más adelante en este mismo capítulo.

A continuación presentamos un ejemplo que utiliza **into** con **group**. El siguiente programa es una versión modificada de **GroupDemo** del ejemplo anterior, que crea una lista de estaciones Web agrupadas por el nombre superior de dominio. En este caso, el resultado inicial es consultado nuevamente por una variable de rango llamada **ws**. Esto da como resultado la eliminación de todos los grupos que tienen menos de tres elementos.

```
// Utiliza into con group.
using System;
using System.Linq;

class IntoDemo {

    static void Main() {

        string[] sitiosweb = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                               "hsNameD.com", "hsNameE.org", "hsNameF.org",
                               "hsNameG.tv", "hsNameH.net", "hsNameI.tv" };

        // Crea un query que agrupa sitios Web por nombre superior de dominio,
        // pero selecciona sólo aquellos grupos que tienen más de dos miembros.
        // Aquí, ws es la variable de rango sobre el conjunto de grupos
        // regresada cuando se ejecuta la primera mitad del query.
        var DirWeb = from dir in sitiosweb
                     where dir.LastIndexOf(".") != -1
                     group dir by dir.Substring(dir.LastIndexOf("." ))
                     into ws ← Coloca el resultado temporal en ws.
                     where ws.Count() > 2
                     select ws;
                     ↑
                     Filtra ws.

        // Ejecuta el query y muestra los resultados.
        Console.WriteLine("Dominios superiores con más de dos miembros.\n");

        foreach(var sitios in DirWeb) {
            Console.WriteLine("Contenido de " + sitios.Key + " dominio:");
            foreach (var sitio in sitios)
                Console.WriteLine(" " + sitio);
            Console.WriteLine();
        }
    }
}
```

Los datos de salida generados por el programa son:

Dominios superiores con más de dos miembros.

Contenido de .net dominio:

```
hsNameB.net
hsNameC.net
hsNameH.net
```

Como lo muestra el resultado, sólo el grupo **.net** es regresado porque es el único que tiene más de dos elementos.

En el programa, presta especial atención a la secuencia de cláusulas en la consulta:

```
group dir by dir.Substring(dir.LastIndexOf( ". " ))
    into ws
where ws.Count() > 2
select ws;
```

Primero, el resultado de la cláusula **group** se almacena (creando un resultado temporal) y principia un nuevo query, el cual opera sobre los resultados obtenidos. La variable de rango del nuevo query es **ws**. En este punto, **ws** operará sobre cada grupo regresado por la primera consulta. (Opera sobre grupos porque el resultado del primer query es una secuencia de grupos.) A continuación, la cláusula **where** filtra la consulta con el objeto de que el resultado final contenga sólo aquellos grupos que incluyen más de dos miembros. Esta determinación se consigue invocando el método **Count()**, que es un *método de extensión* que es implementado para todos los objetos **IEnumerable<T>**. Regresa la cantidad de elementos en una secuencia. (Aprenderás más sobre métodos de extensión más adelante en este mismo capítulo.) La secuencia resultante de grupos es regresada por la cláusula **select**.

## Utilizar **let** para crear una variable en un query

En una consulta, en ocasiones querrás retener temporalmente un valor. Por ejemplo, tal vez quieras crear una variable enumerable que pueda, en sí misma, ser sujeto a una consulta. O bien, quizás quieras almacenar un valor que será utilizado posteriormente en una cláusula **where**. Cualquiera que sea el propósito, estos tipos de acciones pueden realizarse con el uso de **let**.

La cláusula **let** tiene el siguiente formato general:

**let nombre = expresión**

Aquí, *nombre* es un identificador que es asignado al valor de *expresión*. El tipo de *nombre* es inferido a partir del tipo de la expresión.

A continuación presentamos un ejemplo que muestra cómo puede usarse **let** para crear otra fuente de datos enumerable. La consulta toma un arreglo de cadena como datos. Luego convierte esas cadenas en arreglos **char**. Esto se realiza utilizando otro método **string** llamado **ToCharArray()**, que regresa un arreglo que contiene los caracteres de la cadena. El resultado es asignado a una variable llamada **charArreglo**, que luego es utilizada por otra cláusula **from** para obtener los caracteres individuales de un arreglo. Después, la consulta ordena los caracteres y regresa la secuencia resultante.

```
// Utiliza la cláusula let y una cláusula from anidada.
using System;
using System.Linq;

class LetDemo {

    static void Main() {

        string[] strs = { "alpha", "beta", "gamma" };

        // Crea un query que obtiene los caracteres en las
        // cadenas, los regresa en orden alfabético. Observa el uso
        // de la cláusula from anidada.
        var chrs = from str in strs
                   let chrArreglo = str.ToCharArray() ← chrArreglo hace referencia
   a un arreglo de caracteres
   obtenido de str.
                   from ch in chrArreglo
                   orderby ch
                   select ch;

        Console.WriteLine("Los caracteres individuales en orden alfabético
son:");

        // Ejecuta el query y muestra los resultados.
        foreach(char c in chrs) Console.Write(c + " ");

        Console.WriteLine();
    }
}
```

Los datos generados por el programa son:

```
Los caracteres individuales en orden alfabético son:
a a a a a b e g h l m m p t
```

En el programa observa cómo la cláusula **let** asigna a **chrArreglo** una referencia al arreglo regresado por **str.ToCharArray()**:

```
let chrArreglo = str.ToCharArray()
```

Después de la cláusula **let**, otras cláusulas pueden hacer uso de **chrArreglo**. Aún más, como todos los arreglos en C# implementan **IEnumerable<T>**, **chrArreglo** puede usarse como fuente de datos por una segunda cláusula **from** que esté anidada. Esto es lo que sucede en el ejemplo. Utiliza el **from** anidado para enumerar los caracteres individuales del arreglo, ordenándolos en una secuencia ascendente y regresando el resultado.

También puedes utilizar una cláusula **let** para almacenar un valor no enumerable. Por ejemplo, la que sigue es una manera más eficiente de escribir la consulta mediante el programa **Intodemo** presentado en la sección anterior.

```
var DirWeb = from dir in sitiosweb
             let idx = dir.LastIndexOf(".") ← Se invoca una sola vez LastIndexOf(),
   y se ordena el resultado en idx.
             where idx != -1
```

```

group dir by dir.Substring(idx)
    into ws
where ws.Count() > 2
select ws;

```

En esta versión, el índice de la última ocurrencia de un punto se asigna a **idx**. Luego, este valor es utilizado por **Substring()**. Esto previene que la búsqueda de punto se realice una segunda ocasión.

## Unir dos secuencias con **join**

Cuando trabajas con bases de datos, es muy común que quieras correlacionar datos de dos distintas fuentes. Por ejemplo, una tienda en línea puede tener una base de datos que asocie el nombre de un producto con su número de serie y una segunda base de datos que asocie el número de serie con el estatus de existencia en bodega. Dada esta situación, quizás quieras generar una lista que muestre la existencia del producto por el nombre del mismo en lugar de hacerlo por su número de serie. Puedes hacerlo correlacionando los datos de las dos bases. Tal acción se realiza fácilmente en LINQ a través del uso de la cláusula **join**.

El formato general de **join** se muestra a continuación (en el contexto de la cláusula **from**):

```

from var-rangoA in fuente-datosA
join var-rangoB in fuente-datosB
    on var-rangoA.propiedad equals var-rangoB.propiedad

```

La clave para utilizar **join** es entender que cada fuente de datos debe contener datos en común y que esos datos pueden ser comparados en busca de una igualdad. Así, en el formato general, *fuentedatosA* y *fuentedatosB* deben tener algo en común que pueda ser comparado. Los elementos a comparar se especifican en la sección **on**. De esta manera, cuando *var-rangoA.propiedad* es igual (**equals**) a *var-rangoB.propiedad*, la correlación es exitosa. En esencia, **join** actúa como un filtro, permitiendo el paso sólo de aquellos elementos que comparten un valor común.

Cuando se utiliza **join**, por lo regular la secuencia que se regresa es una composición heterogénea de las dos fuentes de datos. Así, **join** te permite generar una nueva lista que contiene elementos de dos diversas fuentes de datos. Esto te permite organizar los datos de una manera completamente nueva.

El siguiente programa crea una clase llamada **Producto**, que encapsula el nombre de un producto con su número de serie. El programa crea otra clase llamada **EstatusExistencia**, que vincula el número de serie de un producto con una propiedad booleana que indica si hay o no existencias de ese producto. También crea una clase llamada **Temp**, que tiene dos campos, uno **string** y otro **bool**. Los objetos de esta clase almacenarán el resultado del query. La consulta utiliza **join** para producir una lista en la cual el nombre del producto es asociado con su estatus de existencia.

```

// Muestra join en funcionamiento.
using System;
using System.Linq;

// Una clase que vincula el nombre de un producto con su número de serie.
class Producto {

```

```
public string Nombre { get; set; }
public int ProdNum { get; set; }

public Producto(string n, int pnum) {
    Nombre = n;
    ProdNum = pnum;
}

// Una clase que vincula el número del producto con su estatus de
// existencia.
class EstatusExistencia {
    public int ProdNum { get; set; }
    public bool EnExistencia { get; set; }

    public EstatusExistencia(int n, bool b) {
        ProdNum = n;
        EnExistencia = b;
    }
}

// Una clase que encapsula el nombre con su estatus.
class Temp {
    public string Nombre { get; set; }
    public bool EnExistencia { get; set; }

    public Temp(string n, bool b) {
        Nombre = n;
        EnExistencia = b;
    }
}

class JoinDemo {
    static void Main() {

        Producto[] productos = {
            new Producto ("Alicates", 1424),
            new Producto ("Martillo", 7892),
            new Producto ("Llave inglesa", 8534),
            new Producto ("Serrucho", 6411)
        };

        EstatusExistencia[] estatusLista = {
            new EstatusExistencia (1424, true),
            new EstatusExistencia (7892, false),
            new EstatusExistencia (8534, true),
            new EstatusExistencia (6411, true)
        };
    }
}
```

```

// Crea un query que une Producto con EstatusExistencia para
// producir una lista de nombres de productos y su disponibilidad.
// Observa que se produce una secuencia de objetos Temp.
var enExistenciaLista = from producto in productos
                         join elemento in estatusLista
                           on producto.ProdNum equals elemento.ProdNum
                         select new Temp(producto.Nombre, elemento.
EnExistencia);

Console.WriteLine("Producto\tDisponibilidad\n");
// Ejecuta el query y muestra los resultados.
foreach(Temp t in enExistenciaLista)
    Console.WriteLine("{0}\t{1}", t.Nombre, t.EnExistencia);
}
}

```

Regresa un objeto **Temp**  
 que contiene los resultados  
 de la unión.

Une dos listas con base  
 en **ProdNum**.

Los datos generados por el programa son:

| Producto      | Disponibilidad |
|---------------|----------------|
| Alicates      | True           |
| Martillo      | False          |
| Llave inglesa | True           |
| Serrucho      | True           |

Para comprender cómo funciona **join**, analicemos cada línea de la consulta. Comienza de manera normal con la cláusula **from**:

```
var enExistenciaLista = from producto in productos
```

Esta cláusula especifica que **producto** es la variable de rango para la fuente de datos especificada por **productos**. El arreglo **productos** contiene objetos de tipo **Producto**, que encapsulan un nombre y un número de ciertos elementos del inventario.

A continuación viene la cláusula **join**, que se muestra aquí:

```
join elemento in estatusLista
  on producto.ProdNum equals elemento.ProdNum
```

En este bloque se especifica qué **elemento** es la variable de rango de la fuente de datos **estatusLista**. El arreglo **estatusLista** contiene objetos de tipo **EstatusExistencia**, que vincula un número de producto con su estatus. Así, **productos** y **estatusLista** tienen una propiedad en común, el número del producto. Esto es utilizado por la porción **on>equals** de la cláusula **join** para describir la correlación. De esta manera, **join** encuentra elementos coincidentes de las dos fuentes de datos cuando sus números de producto son iguales.

Finalmente, la cláusula **select** regresa un objeto **Temp** que contiene el nombre del producto junto con su estatus de existencia:

```
select new Temp(producto.Nombre, elemento.EnExistencia);
```

Por lo mismo, la secuencia obtenida por la consulta consiste en objetos **Temp**.

Aunque el ejemplo anterior es muy directo, **join** soporta operaciones mucho más sofisticadas. Por ejemplo, puedes utilizar **into** con **join** para crear una *conjunción de grupos*, la cual crea un resultado que consiste en un elemento de la primera secuencia y un grupo de todos los elementos coincidentes de la segunda secuencia. (Verás un ejemplo de ello más adelante en este mismo capítulo.) En general, el tiempo y esfuerzo ocupados para dominar **join** valen la pena porque te ofrece la posibilidad de reorganizar tus datos en tiempo de ejecución. Ésta es una capacidad muy poderosa. Es todavía más potente con el uso de tipos anónimos, descritos en la siguiente sección.

## Tipos anónimos e inicializadores de objetos

La versión 3.0 de C# añade una característica llamada *tipo anónimo* relacionada directamente con LINQ. Como su nombre indica, un tipo anónimo es una clase que no tiene nombre. Su uso principal es crear un objeto regresado por la cláusula **select**. A menudo, el resultado de una consulta es una secuencia de objetos que es una composición de dos (o más) fuentes de datos (como en el caso de **join**) o bien incluye un subconjunto de miembros de una fuente de datos. En cualquier caso, el tipo que es regresado se requiere sólo para efectos de ejecución del query, pero no es utilizado en ninguna otra parte del programa. En este caso, el uso de un tipo anónimo elimina la necesidad de declarar una clase que sólo será utilizada para almacenar el resultado del query.

Un tipo anónimo se crea con el uso del siguiente formato:

```
new {nombreA = valorA, nombreB = valorB, ...}
```

Aquí, los nombres especifican identificadores que son transformados en propiedades sólo-lectura, que son inicializadas por los valores. Por ejemplo,

```
new { Cuenta = 10, Max = 100, Min = 0 }
```

Esto crea un tipo de clase que tiene tres propiedades sólo-lectura públicas: **Cuenta**, **Max** y **Min**, a las cuales se les asignan los valores 10, 100 y 0, respectivamente. Estas propiedades pueden ser referidas por nombre o por código. La sintaxis recibe el nombre de *inicialización de objeto*. Es otra característica nueva de C#. Proporciona un medio para inicializar un objeto sin invocar explícitamente un constructor. Esto es necesario en el caso de los tipos anónimos porque no existe una manera de invocar explícitamente un constructor. (Recuerda que los constructores tienen el mismo nombre de su clase. En el caso de una clase anónima, no hay un nombre. Así que, ¿cómo podrías invocar un constructor?)

Como un tipo anónimo carece de nombre, debes utilizar un tipo implícito de variable para hacer la referencia. Esto permite al compilador inferir el tipo apropiado. Por ejemplo,

```
var miOb = new { Cuenta = 10, Max = 100, Min = 0 }
```

crea una variable llamada **miObj** a la que le es asignada una referencia al objeto creado por la expresión de tipo anónimo. Esto significa que las siguientes declaraciones son legales:

```
Console.WriteLine("Cuenta es" + miOb.Cuenta);  
if(i <= miOb.Max && i >= miOb.Min) // ...
```

Recuerda, cuando se crea un tipo anónimo, los identificadores que específicas se convierten en propiedades públicas sólo-lectura. Así, pueden usarse por otras partes de tu código.

Aunque se utiliza el término *tipo anónimo*, ¡no es completamente verdadero! El tipo es anónimo en relación contigo, con el programador. Sin embargo, el compilador sí le da un nombre interno. Así, los tipos anónimos no violan las rígidas reglas de verificación de tipos de C#.

Para comprender completamente el valor de los tipos anónimos, considera la siguiente versión del programa anterior que muestra el uso de **join**. Recuerda que en la versión previa, una clase llamada **Temp** era necesaria para encapsular el resultado de **join**. Con el uso de tipos anónimos, esta clase “marcadora de posición” ya no es necesaria y ya no desordena el código fuente de tu programa. Los datos de salida que genera esta versión son los mismos que en la anterior.

```
// Utiliza un tipo anónimo para mejorar el programa que muestra join.
using System;
using System.Linq;

// Una clase que vincula el nombre de un producto con su número de serie.
class Producto {
    public string Nombre { get; set; }
    public int ProdNum { get; set; }

    public Producto(string n, int pnum) {
        Nombre = n;
        ProdNum = pnum;
    }
}

// Una clase que vincula un número de producto con su estatus de
// existencia.
class EstatusExistencia {
    public int ProdNum { get; set; }
    public bool EnExistencia { get; set; }

    public EstatusExistencia (int n, bool b) {
        ProdNum = n;
        EnExistencia = b;
    }
}

class TipoAnonDemo {
    static void Main() {

        Producto[] productos = {
            new Producto ("Alicates", 1424),
            new Producto ("Martillo", 7892),
            new Producto ("Llave inglesa", 8534),
            new Producto ("Serrucho", 6411)
        };
    }
}
```

```

        EstatusExistencia[] estatusLista = {
            new EstatusExistencia (1424, true),
            new EstatusExistencia (7892, false),
            new EstatusExistencia (8534, true),
            new EstatusExistencia (6411, true)
        };

        // Crea un query que une Producto con EstatusExistencia para
        // producir una lista de nombres de productos y su disponibilidad.
        // Ahora se utiliza un tipo anónimo.
        var enExistenciaLista = from producto in productos
                                join elemento in estatusLista
                                on producto.ProdNum equals elemento.ProdNum
                                select new { Nombre = producto.Nombre,
  EnExistencia = elemento.
  EnExistencia };

        Console.WriteLine("Producto\tDisponible\n");
    }

    // Ejecuta el query y muestra los resultados.
    foreach(var t in enExistenciaLista)
        Console.WriteLine("{0}\t{1}", t.Nombre, t.EnExistencia);
}
}

```

↑  
Regresa un tipo anónimo.

Pon especial atención a la cláusula **select**:

```
select new { Nombre = producto.Nombre,
            EnExistencia = elemento.EnExistencia };
```

Regresa un objeto de tipo anónimo que tiene dos propiedades sólo-lectura, **Nombre** y **EnExistencia**. A ellos se les asignan los valores especificados por el nombre y la disponibilidad del producto. Gracias al tipo anónimo, ya no se requiere la clase **Temp**.

Otro punto: observa el loop **foreach**. Ahora utiliza **var** para declarar la variable de reiteración. Esto es necesario porque el tipo del objeto contenido en **enExistenciaLista** no tiene nombre. Esta situación es una de las razones por las que C# 3.0 añadió las variables de tipo implícito. Son necesarias para soportar los tipos anónimos.

Antes de continuar, hay otro aspecto de los tipos anónimos que merece mención. En algunos casos, incluyendo el que acabas de ver, puedes simplificar la sintaxis del tipo anónimo utilizando un *inicializador de proyección*. En este caso, simplemente especificas el nombre del inicializador en sí. El nombre se convierte automáticamente en el nombre de la propiedad. Por ejemplo, a continuación aparece otra manera de presentar la cláusula **select** utilizada en el programa anterior:

```
select new { producto.Nombre, elemento.EnExistencia };
```

Aquí, los nombres de propiedad siguen siendo **Nombre** y **EnExistencia**, como antes. El compilador automáticamente “proyecta” los identificadores **Nombre** y **EnExistencia**, haciéndolos los nombres de propiedad del tipo anónimo. Además, como antes, a las propiedades se les asignan los valores especificados por **producto.Nombre** y **elemento.EnExistencia**.

## Pregunta al experto

**P:** La sintaxis de inicialización del objeto utilizada por un tipo anónimo ¿también puede usarse para tipos con nombre?

**R:** ¡Sí! La sintaxis de inicialización de objeto puede utilizarse en tipos con nombre. Por ejemplo, dada la siguiente clase:

```
class MiClase {
    public int Alpha { get; set; }
    public int Beta { get; set; }
}
```

La siguiente declaración es legal:

```
var MiOb = new MiClase { Alpha = 10, Beta = 20 };
```

Después de que se ejecuta esta declaración, la línea

```
Console.WriteLine("Alpha: {0}, Beta {1}", miOb.Alpha, miOb.Beta);
```

muestra

```
Alpha: 10, Beta 20
```

Aunque los inicializadores de objetos puedan ser utilizados con clases con nombre, su uso primario es con tipos anónimos. Por lo mismo, normalmente deberás invocar explícitamente un constructor cuando trabajes con clases que tienen nombre.

## Crear una conjunción de grupo

Como se explicó anteriormente, puedes utilizar **into** con **join** para crear una *conjunción de grupo*, que crea una secuencia en la cual cada entrada del resultado consiste en un elemento de la primera secuencia y un grupo de todos los elementos coincidentes de la segunda secuencia. No habíamos presentado un ejemplo de ello porque por lo regular una conjunción de grupo requiere el uso de un tipo anónimo. Ahora que hemos explicado el uso de los tipos anónimos, podemos ver un ejemplo de conjunción de grupo sencilla.

El siguiente ejemplo utiliza una conjunción de grupo para crear una lista en la cual varios transportes, como automóviles, botes y aeroplanos, se organizan por su categoría general de transporte, que son tierra, mar o aire. Primero, el programa crea una clase llamada **Transporte** que vincula un tipo de transporte con su clasificación. Dentro de **Main()**, crea dos secuencias. La primera es un arreglo de cadenas que contiene los nombres de los medios generales por los que uno se transporta: tierra, mar y aire. La segunda es un arreglo de **Transporte** que encapsula varios medios

de desplazamiento. Luego utiliza una conjunción de grupo para producir una lista de transportes organizados por categoría.

```
// Muestra una conjunción de grupo sencilla.
using System;
using System.Linq;

// Esta clase vincula el nombre de un medio de transporte, como Tren,
// con su clasificación general, como tierra, mar o aire.
class Transporte {
    public string Nombre { get; set; }
    public string Técnica { get; set; }

    public Transporte (string n, string t) {
        Nombre = n;
        Técnica = t;
    }
}

class ConjunGrupoDemo {
    static void Main() {

        // Un arreglo de clasificación de transporte.
        string[] tiposViaje = {
            "Aire",
            "Mar",
            "Tierra",
        };

        // Un arreglo de transportes.
        Transporte[] medios = {
            new Transporte ("Bicicleta", "Tierra"),
            new Transporte ("Globo", "Aire"),
            new Transporte ("Bote", "Mar"),
            new Transporte ("Jet", "Aire"),
            new Transporte ("Canoa", "Mar"),
            new Transporte ("Biplano", "Aire"),
            new Transporte ("Carro", "Tierra"),
            new Transporte ("Carguero", "Mar"),
            new Transporte ("Tren", "Tierra")
        };

        // Crea un query que utiliza conjunción de grupo para producir
        // una lista de nombres de elementos e ID organizados por categoría.
        var conTec = from técnica in tiposViaje
                    join trans in medios
                    on técnica equals trans.Técnica
                    into lst
                    select new { Tec = técnica, Tlista = lst };
    }
}
```

Crea una  
conjunción  
de grupo.

```
// Ejecuta el query y muestra los resultados.
foreach(var t in conTec) {
    Console.WriteLine("{0} transportación incluye:", t.Tec);

    foreach(var m in t.Tlista)
        Console.WriteLine(" " + m.Nombre);

    Console.WriteLine();
}

}
```

Los resultados generados por el programa son:

Aire transportación incluye:

Globo  
Jet  
Biplano

Mar transportación incluye:

Bote  
Canoa  
Carguero

Tierra transportación incluye:

Bicicleta  
Carro  
Tren

La parte clave del programa es, por supuesto, la consulta, la cual se muestra a continuación:

```
var conTec = from técnica in tiposViaje
              join trans in medios
              on técnica equals trans.Técnica
              into lst
              select new { Tec = técnica, Tlista = lst };
```

He aquí cómo funciona. La declaración **from** utiliza **técnica** para recorrer el arreglo **tiposViaje**. Recuerda que **tiposViaje** contiene un arreglo de la clasificación general de transporte: aire, tierra y mar. La cláusula **join** conjunta cada tipo de viaje con aquellos medios de transporte que utilizan los respectivos tipos de viaje. Por ejemplo, el tipo tierra se conjunta con Bicicleta, Auto y Tren. Sin embargo, debido a la cláusula **into**, por cada tipo de viaje, **join** produce una lista de transportes que utilizan ese tipo. Esta lista está representada por **lst**. Finalmente, **select** regresa un tipo anónimo que encapsula cada valor de **técnica** (el tipo de viaje) con una lista de los medios de transporte. Por eso se requieren dos loop **foreach** para mostrar el resultado del query. El loop externo obtiene un objeto que contiene el nombre del tipo de viaje y una lista de los medios de trasporte para ese tipo. El loop interno muestra los transportes individuales.

Existen muchas opciones y matices asociados con la conjunción de grupo. De hecho, este tipo de conjunción es una de las técnicas más sofisticadas del query, y es una de las características que querrás explorar con mayor profundidad. Una de las mejores maneras de hacerlo es experimentar con ejemplos sencillos, asegurándote de que entiendes perfectamente lo que está ocurriendo en cada paso.

## Los métodos de query y las expresiones lambda

La sintaxis de consulta descrita en las secciones anteriores, probablemente será la manera como escribas la mayoría de consultas en C#. Es conveniente, poderosa y compacta. Sin embargo, no es la única manera de escribir un query. La otra manera de hacerlo es utilizando *métodos de query*. Estos métodos pueden ser invocados en cualquier objeto enumerable, como un arreglo. Muchos de los métodos de consulta requieren el uso de otra nueva característica de C# 3.0: la expresión lambda. Como los métodos de consulta y las expresiones lambda están entrelazadas, ambas se presentan a continuación.

### Los métodos de query básicos

Los métodos de query están definidos por **System.Linq.Enumerable** y son implementados como métodos de extensión que expanden la funcionalidad de **IEnumerable<T>**. (Los métodos de query son también definidos por **System.Linq.Queryable**, que extiende la funcionalidad de **IQueryable<T>**, pero esta interfaz no se utiliza en este capítulo.) Un método de extensión añade funcionalidad a otra clase, pero sin el uso de la herencia. El soporte a los métodos de extensión fue añadido en la versión 3.0 de C#, y lo veremos más de cerca en este mismo capítulo. Por el momento, es suficiente comprender que los métodos de consulta pueden ser invocados sólo en objetos que implementan **IEnumerable<T>**.

La clase **Enumerable** proporciona muchos métodos de consulta, pero en el centro están aquellos que corresponden a las palabras clave del query descritas anteriormente. Estos métodos se muestran a continuación, junto con las palabras clave con las que se relacionan. Debes comprender que estos métodos tienen formatos sobrecargados y ahora sólo presentamos su formato más sencillo. Sin embargo, éste será el formato que utilizarás regularmente.

| Palabra clave query | Método de query equivalente           |
|---------------------|---------------------------------------|
| select              | Select(arg)                           |
| where               | Where(arg)                            |
| order               | OrderBy(arg) u OrderByDescending(arg) |
| join                | Join(sec2, clave1, clave2, resultado) |
| group               | GroupBy(arg)                          |

Con excepción de **Join( )**, los demás métodos toman un argumento, *arg*, que es un objeto de tipo **Func<T, TResult>**, como un parámetro. Éste es un tipo delegado definido por LINQ. Se declara así:

```
delegate TResult Func<T, TResult> (T arg)
```

Aquí, **TResult** especifica el resultado del delegado y **T** especifica el tipo de parámetro. En los métodos de query, *arg* determina la acción que toma el método de consulta. Por ejemplo, en el caso de **Where( )**, *arg* determina la manera en que la consulta filtra los datos. Cada uno de estos métodos query regresa un objeto enumerable. Así, el resultado de uno puede utilizarse para ejecutar o invocar otro, permitiendo que los métodos se encadenen entre sí.

El método **Join( )** toma cuatro argumentos. El primero es una referencia a la segunda secuencia que será unida. La primera secuencia es aquella en la cual se invoca **Join( )**. El selector clave para la primera secuencia es transmitido por *clave1*, y el selector clave para la segunda secuencia se transmite por *clave2*. El resultado de la unión es descrito por *resultado*. El tipo de *clave1* es **Func<TOOuter, TKey>**, y el tipo de *clave2* es **Func<TInner, TKey>**. El argumento *resultado* es de tipo **Func<TOOuter, TInner, TResult>**. Aquí, **TOOuter** es el tipo de elemento de la secuencia invocante, **TInner** es el tipo de elemento de la secuencia transmitida y **TResult** es el tipo de los elementos resultantes. Se regresa un objeto enumerable que contiene el resultado de la unión.

Antes de ver los ejemplos que utilizan los métodos de query, es necesario que conozcas las expresiones lambda.

## Expresiones lambda

Aunque el argumento para un método de query como **Where( )** debe ser de tipo **Func<T, TResult>**, no necesita ser un método explícitamente declarado. De hecho, la mayoría de las ocasiones no lo será. En vez de ello, por lo regular utilizarás una *expresión lambda*. Una expresión lambda es una nueva característica sintáctica proporcionada por C# 3.0. Ofrece una manera sencilla pero poderosa para definir lo que es, en esencia, un método anónimo. El compilador de C# convierte automáticamente una expresión lambda en un formato que puede ser transmitido a un parámetro **Func<T, TResult>**.

Aunque examinaremos con gran detalle las expresiones lambda más tarde en esta sección, ahora presentaremos una panorámica general. Todas las expresiones lambda utilizan el nuevo operador *lambda*, que es **=>**. Este operador divide una expresión lambda en dos partes. A la izquierda se especifica el parámetro (o parámetros) de entrada. A la derecha una de dos cosas: una expresión o un bloque de declaraciones. Si el lado derecho es una expresión, entonces se crea una *expresión lambda*. Si el lado derecho es un bloque de declaraciones, entonces se trata de una *declaración lambda*. Para propósitos de esta sección, sólo utilizaremos expresiones lambda.

En una expresión lambda, la expresión del lado derecho del operador **=>** actúa sobre el parámetro (o parámetros) especificados en el lado izquierdo. El resultado de la expresión se convierte en el resultado del operador lambda. He aquí el formato general de una expresión lambda que toma sólo un parámetro:

*param => expr*

Cuando se requiere más de un parámetro, se utiliza el siguiente formato:

*(lista-param) => expr*

Por tanto, cuando se requieren dos o más parámetros, deben ir encerrados entre paréntesis. Si no se necesitan los parámetros, entonces se debe utilizar un juego de paréntesis vacíos.

He aquí una expresión lambda sencilla:

```
n => n > 0
```

Para toda **n**, esta expresión determina si **n** es mayor que cero y regresa el resultado. He aquí otro ejemplo:

```
cuenta => cuenta + 2
```

En este caso, el resultado es el valor de **cuenta** al que se le suma 2.

## Crear consultas utilizando los métodos de query

Utilizando los métodos de consulta en conjunción con las expresiones lambda, es posible crear queries que no utilizan la sintaxis de query propia de C#. En lugar de ello, se invocan los métodos de query. Comencemos con un ejemplo sencillo. Es una versión modificada del primer ejemplo de este capítulo, por lo que utiliza invocaciones a **Where()** y **Select()** en lugar de las palabras clave de query.

```
// Utiliza los métodos query para crear una consulta sencilla.  
// Este programa es una versión modificada del primer programa  
// presentado en este capítulo.  
using System;  
using System.Linq;  
  
class SimpQuery {  
    static void Main() {  
  
        int[] nums = { 1, -2, 3, 0, -4, 5 };  
  
        // Utiliza Where() y Select() para crear un query sencillo.  
        var posNums = nums.Where(n => n > 0).Select(r => r); ← Utiliza métodos  
        // ejecuta el query y muestra los resultados.  
        foreach(int i in posNums) Console.WriteLine(i);  
    }  
}
```

Los datos de salida, mostrados aquí, son los mismos que la versión original:

```
Los valores positivos en nums:  
1  
3  
5
```

En el programa, presta especial atención a esta línea:

```
var posNums = nums.Where(n => n > 0).Select(r => r);
```

Crea un query llamado **posNums** para crear una secuencia de valores positivos en **nums**. Lo hace utilizando el método **Where()** para filtrar los valores y **Select()** para seleccionarlos. El método

**Where( )** puede ser invocado en **nums** porque todos los arreglos implementan **IEnumerable<T>**, que soporta las extensiones de métodos query.

Técnicamente, en el ejemplo anterior el método **Select( )** no es necesario, porque en este caso sencillo, la secuencia regresada por **Where( )** ya contiene el resultado. Sin embargo, puedes utilizar un criterio de selección más sofisticado, como lo hiciste con la sintaxis del query. Por ejemplo, el siguiente query regresa los valores positivos en **nums** incrementados en orden de magnitud:

```
var posNums = nums.Where(n => n > 0).Select(r => r * 10);
```

Como era de esperarse, puedes concatenar otras operaciones entre sí. Por ejemplo, la siguiente consulta selecciona los valores positivos, los organiza en orden descendente y regresa la secuencia resultante:

```
var posNums = nums.Where(n => n > 0).OrderByDescending(j => j);
```

Aquí, la expresión **j => j** especifica que el orden es dependiente del parámetro de entrada, el cual es un elemento de la secuencia obtenida por **Where( )**.

A continuación presentamos un ejemplo para mostrar el funcionamiento del método **GroupBy( )**. Es una versión modificada del ejemplo **group** mostrado anteriormente.

```
// Muestra el funcionamiento del método de query GroupBy().
// Es una versión modificada del ejemplo anterior que utiliza
// la sintaxis de query.
using System;
using System.Linq;

class GroupByDemo {

    static void Main() {

        string[] sitiosweb = { "hsNameA.com", "hsNameB.net", "hsNameC.net",
                               "hsNameD.com", "hsNameE.org", "hsNameF.org",
                               "hsNameG.tv", "hsNameH.net", "hsNameI.tv" };

        // Utiliza métodos query para agrupar sitios web por su nombre
        // superior de dominio.
        var dirWeb = sitiosweb.Where(w => w.LastIndexOf(".") != 1).
            GroupBy(x => x.Substring(x.LastIndexOf(".", x.Length)));

        // Ejecuta el query y muestra los resultados.
        foreach(var sitios in dirWeb) {
            Console.WriteLine("Sitios Web agrupados por " + sitios.Key);
            foreach(var sitio in sitios)
                Console.WriteLine(" " + sitio);
            Console.WriteLine();
        }
    }
}
```

Esta versión produce el mismo resultado que el programa original. La única diferencia es la manera como se crea el query. En esta versión se utilizan métodos de consulta.

He aquí otro ejemplo. Recuerda el query **join** utilizando en el ejemplo **JoinDemo** mostrado anteriormente:

```
var enExistenciaLista = from producto in productos
                        join elemento in estatusLista
                            on producto.ProdNum equals elemento.ProdNum
                        select new Temp (producto.Nombre, elemento.
   EnExistencia);
```

Esta consulta produce una secuencia que contiene objetos que encapsulan el nombre y el estatus en existencia de un producto en inventario. Esta información es la síntesis que resulta de unir dos listas llamadas **productos** y **estatusLista**. La siguiente es una nueva versión de este query que utiliza el método **Join()** en lugar de la sintaxis query de C#:

```
// Usa Join() para producir una lista con el nombre del producto y su
estatus de existencia.
var enExistenciaLista = productos.Join (estatusLista,
   k1 => k1.ProdNum,
   k2 => k2.ProdNum,
   (k1, k2) => new Temp(k1.Nombre, k2.EnExistencia));
```

Aunque esta versión utiliza una clase con nombre llamada **Temp** para contener el objeto resultante, en lugar de ello se puede utilizar un tipo anónimo. Este enfoque se muestra a continuación:

```
var enExistenciaLista = productos.Join (estatusLista,
   k1 => k1.ProdNum,
   k2 => k2.ProdNum,
   (k1, k2) => new { k1.Nombre, k2.EnExistencia} );
```

### Pregunta al experto

**P:** He leído el término **árbol de expresión** utilizado en el contexto de las expresiones lambda. ¿Qué significa?

**R:** Un árbol de expresión es la representación de una expresión lambda como dato. Así, un árbol de expresión, en sí, no puede ser ejecutado. Sin embargo, puede ser convertido en un formato ejecutable. Los árboles de expresión están encapsulados por la clase **System.Linq.Expressions.Expression<T>**. Los árboles de expresión son útiles en situaciones en las cuales será ejecutado un query por una entidad externa al programa, como una base de datos que utilice SQL. Al representar la consulta como datos, se puede convertir en un formato entendible para la base de datos. Este proceso es utilizado por la característica LINQ-a-SQL proporcionada por Visual C#, por ejemplo. Así, los árboles de expresión ayudan a C# a soportar diversas fuentes de datos.

Puedes obtener un formato ejecutable de un árbol de expresión invocando el método **Compile()** definido por **Expression**. Regresa una referencia que puede asignarse a un delegado para luego ser ejecutado. Los árboles de expresión tienen una restricción clave: sólo pueden representar expresiones lambda. No pueden utilizarse para representar declaraciones lambda. Aunque en este capítulo no utilizaremos explícitamente árboles de expresión, son algo que encontrarás muy interesante para investigar por tu cuenta.

## Más extensiones de métodos relacionados con query

Además de los métodos que corresponden a las palabras clave del query, existen otros métodos relacionados con él, definidos por **IEnumerable<T>** a través de **Enumerable**. A continuación presentamos una muestra de los métodos más comúnmente utilizados. Como muchos de los métodos son sobrecargados, sólo se muestra su formato general.

| Método         | Descripción                                                                                          |
|----------------|------------------------------------------------------------------------------------------------------|
| All(condición) | Regresa un valor true si todos los elementos en una secuencia satisfacen una condición especificada. |
| Any(condición) | Regresa un valor true si cualquier elemento en una secuencia satisface una condición especificada.   |
| Average( )     | Regresa el promedio de los valores en una secuencia numérica.                                        |
| Contains(obj)  | Regresa un valor true si la secuencia contiene el objeto especificado.                               |
| Count( )       | Regresa la longitud de la secuencia. Ésta es la cantidad de elementos que contiene.                  |
| First( )       | Regresa el primer elemento de la secuencia.                                                          |
| Last( )        | Regresa el último elemento de la secuencia.                                                          |
| Max( )         | Regresa el valor máximo en una secuencia.                                                            |
| Min( )         | Regresa el valor mínimo en una secuencia.                                                            |
| Sum( )         | Regresa la sumatoria de los valores en una secuencia numérica.                                       |

Ya has visto **Count( )** en acción en este capítulo. He aquí un programa que muestra a los demás en acción:

```
// Usa varios de los métodos de extensión definidos por Enumerable.
using System;
using System.Linq;

class ExtMetodos {
    static void Main() {
        int[] nums = { 3, 1, 2, 5, 4 };

        Console.WriteLine("El valor mínimo es " + nums.Min());
        Console.WriteLine("El valor máximo es " + nums.Max());

        Console.WriteLine("El primer valor es " + nums.First());
        Console.WriteLine("El último valor es " + nums.Last());

        Console.WriteLine("La sumatoria es " + nums.Sum());
        Console.WriteLine("El promedio es " + nums.Average());

        if(nums.All(n => n > 0))
            Console.WriteLine("Todos los valores son mayores que cero.");

        if(nums.Any(n => (n % 2) == 0))
            Console.WriteLine("Por lo menos un valor es par.");
    }
}
```

```
if(nums.Contains(3))
    Console.WriteLine("El arreglo contiene el valor 3.");
}
}
```

Los datos generados por el programa son:

```
El valor mínimo es 1
El valor máximo es 5
El primer valor es 3
El último valor es 4
La sumatoria es 15
El promedio es 3
Todos los valores son mayores que cero.
Por lo menos un valor es par.
El arreglo contiene el valor 3.
```

También puedes utilizar estos métodos de extensión dentro de una consulta basada en la sintaxis de query de C#. Por ejemplo, el siguiente programa utiliza **Average()** para obtener una secuencia que contenga sólo los valores menores al promedio de los valores del arreglo.

```
// Usa Average() dentro de la sintaxis de query.
using System;
using System.Linq;

class ExtMet2 {
    static void Main() {

        int[] nums = { 1, 2, 4, 8, 6, 9, 10, 3, 6, 7 };

        var ltAvg = from n in nums
                    let x = nums.Average()
                    where n < x
                    select n; ← Utiliza un método de consulta
                           con la sintaxis de query.

        Console.WriteLine("El promedio es " + nums.Average());

        Console.WriteLine("Estos valores son menores que el promedio:");

        // Ejecuta el query y muestra el resultado.
        foreach(int i in ltAvg) Console.WriteLine(i);
    }
}
```

Los datos generados por el programa son:

```
El promedio es 5.6
Estos valores son menores que el promedio:
1
2
4
3
```

## Aplazamiento vs. inmediatez en la ejecución del query

Antes de seguir adelante, hay un concepto más que necesita ser presentado. En LINQ, los queries tienen dos diferentes modos de ejecución: inmediato y diferido. En general, una consulta define un conjunto de reglas que no entran en acción hasta que una declaración **foreach** se ejecuta. A esto se le llama *ejecución diferida*.

Sin embargo, si utilizas uno de los métodos de extensión que no producen una secuencia de resultados, entonces es necesario que la consulta se ejecute para obtener el resultado. Por ejemplo, considera el método **Count()**. Para que **Count()** regrese la cantidad de elementos en la secuencia, es necesario que la consulta se ejecute, y esto se hace automáticamente cuando se invoca **Count()**. En este caso sucede una *ejecución inmediata*, el query se ejecuta automáticamente para obtener el resultado. Por ello, aunque no utilices explícitamente el query en un loop **foreach**, éste se ejecuta de cualquier manera.

He aquí un ejemplo. Obtiene la cantidad de elementos positivos en una secuencia.

```
// Utiliza la ejecución inmediata.
using System;
using System.Linq;

class EjecInmediata {
    static void Main() {
        int[] nums = { 1, -2, 3, 0, -4, 5 };

        // Crea un query que obtiene la cantidad de valores
        // positivos en nums.
        int longitud = (from n in nums
                        where n > 0
                        select n).Count(); ← Este query se ejecuta de inmediato,
                                // regresando la cantidad de elementos
                                // solicitada.

        Console.WriteLine("La cantidad de valores positivos en nums: " +
            longitud);
    }
}
```

Los datos generados por el programa son:

```
La cantidad de valores positivos en nums: 3
```

Observa en el programa que no se especifica explícitamente un loop **foreach**; la consulta se ejecuta de manera automática debido a la invocación de **Count()**.

Como dato de interés diremos que el query en el programa anterior también pudo ser escrito así:

```
var posNums = from n in nums
               where n > 0
               select n;

int longitud = posNums.Count(); // el query se ejecuta aquí.
```

En este caso, **Count()** es invocado en la variable de consulta. En ese punto, la consulta se ejecuta para obtener el conteo.

## Pregunta al experto

**P:** ¿Por qué C# tiene dos modos de crear queries, la sintaxis de query y los métodos de query?

**R:** De hecho, haciendo a un lado la sintaxis involucrada, sólo tiene un modo. ¿Cómo es esto? ¡Porque la sintaxis de query se compila en las invocaciones a sus métodos! De esta manera, cuando escribes algo como:

```
where x < 10
```

el compilador lo traduce en

```
Where (x => x < 10)
```

Así, las dos maneras de crear un query conducen, en el último de los casos, al mismo lugar.

Esto nos lleva a una segunda pregunta: ¿qué medio se debe utilizar en un programa C#? Por lo regular querrás utilizar la sintaxis de query. Es un medio limpio y completamente integrado al lenguaje C#.

## Una mirada más cercana a la extensión de métodos

Los métodos de extensión proporcionan un medio a través del cual se puede añadir funcionalidad a la clase sin necesidad de utilizar el mecanismo normal de herencia. Aunque no crearás muy seguido tus propios métodos de extensión (porque el mecanismo de herencia ofrece mejores soluciones en la mayoría de los casos), aún así resulta importante que comprendas cómo funcionan, debido a su importancia integral con LINQ.

Un método de extensión es un método estático que debe estar contenido dentro de una clase estática y no genérica. El tipo de su primer parámetro determina el tipo de los objetos en los cuales puede ser invocada la extensión del método. Más aún, el primer parámetro debe ser modificado por la palabra clave **this**. El objeto en el cual el método es invocado se transmite automáticamente

al primer parámetro. No es transmitido de manera explícita en la lista de argumentos. Un punto clave es que aun cuando un método de extensión es declarado **static**, de cualquier manera puede ser invocado en un objeto, como si fuera un método de instancia.

He aquí el formato general de un método de extensión:

```
static tipo-ret nombre (this invoca-en-tipo ob, lista-param)
```

Por supuesto, si no existen otros argumentos además de los que se transmiten implícitamente a *ob*, la *lista-param* quedará vacía. Recuerda que el primer parámetro es transmitido automáticamente al objeto en el cual el método es invocado. En general, un método de extensión será un miembro público de su clase.

He aquí un ejemplo que crea tres métodos de extensión sencillos:

```
// Crea y utiliza algunos métodos de extensión.
using System;

static class MiExtMets {

    // Regresa el recíproco de un double.
    public static double Recíproco(this double v) { ←
        return 1.0 / v;
    }

    // Invierte las letras mayúsculas y minúsculas en un string.
    // y regresa el resultado.
    public static string RevMay this string str) { ←
        string temp = "";

        foreach(char ch in str) {
            if(Char.IsLower(ch)) temp += Char.ToUpper(ch);
            else temp += Char.ToLower(ch);
        }
        return temp;
    }

    // Regresa el valor absoluto de n / d.
    public static double AbsDivideBy(this double n, double d) { ←
        return Math.Abs(n / d);
    }
}

class ExtDemo {
    static void Main() {
        double val = 8.0;
        string str = "Alpha Beta Gamma";

        // Invoca el método de extensión Recíproco().
        Console.WriteLine("Recíproco de {0} es {1}",
                           val, val.Recíproco());

        // Invoca el método de extensión RevMay().
    }
}
```

Métodos de extensión. Observa el uso de **this**.

```
Console.WriteLine(str + " después de la inversión es " +
                  str.RevMay());
// Usa AbsDivideBy();
Console.WriteLine("Resultado de val.AbsDivideBy(-2): " +
                  val.AbsDivideBy(-2));
}
}
```

Los resultados generados por el programa son:

```
Recíproco de 8 es 0.125
Alpha Beta Gamma después de la inversión es aLPHA bETA gAMMA
Resultado de val.AbsDivideBy(-2): 4
```

Observa en el programa que cada método de extensión está contenido en una clase estática llamada **MiExtMets**. Como se explicó, un método de extensión debe ser declarado dentro de una clase estática. Más aún, esta clase debe ser de alcance general con el fin de que el método de extensión que contiene pueda utilizarse. (Por esta razón necesitas incluir la nomenclatura **System.Linq** cuando utilizas LINQ.) A continuación, observa la invocación a los métodos de extensión. Son invocados en un objeto de la misma manera como se invoca un método de instancia. La principal diferencia consiste en que el objeto invocante es transmitido al primer parámetro del método de extensión. Por lo mismo, cuando se ejecuta la expresión

```
val.AbsDivideBy(-2))
```

**val** es transmitido al parámetro **n** de **AbsDivideBy()** y **-2** es transmitido al parámetro **d**.

Como punto de interés, dado que los métodos **Recíproco()** y **AbsDivideBy()** están definidos para el tipo **double**, es legal invocarlos en una literal **double**, como se muestra aquí:

```
8.0.Recíproco()
8.0.AbsDivideBy(-1)
```

Más aún, **RevMay()** puede invocarse así:

```
"AbCDe".RevMay()
```

En este caso, regresa la versión invertida de la literal de cadena.

## Una mirada más cercana a las expresiones lambda

Aunque el uso principal de las expresiones lambda es con LINQ, son una característica que puede utilizarse con otros aspectos de C#. La razón es que las expresiones lambda crean otro tipo de función anónima. (El otro tipo de función anónima es el método anónimo, descrito con anterioridad en este libro.) Así, una expresión lambda puede ser asignada (o transmitida) a un delegado. Dado que una expresión lambda es más eficiente que su método anónimo equivalente, las expresiones lambda son ahora la técnica más recomendada para la mayoría de los casos.

Como se mencionó anteriormente, C# soporta dos tipos de expresión lambda. La primera es llamada *expresión lambda* y es la que has venido utilizando hasta este momento. El cuerpo de una expresión lambda es independiente, es decir, no está encerrada entre llaves. El segundo tipo es la *declaración lambda*. En una declaración lambda, el cuerpo está encerrado entre llaves.

Una declaración lambda puede incluir otras declaraciones C#, como loops, invocaciones a métodos y declaraciones **if**. Los dos tipos de lambda son examinados a continuación.

## Expresión lambda

El cuerpo de una expresión lambda consiste únicamente en la expresión en el lado derecho del operador **=>**. Así, cualquiera que sea la acción que realiza la expresión lambda, ésta debe ocurrir dentro de una sola expresión. Las expresiones lambda son típicamente utilizadas con queries, como lo han mostrado los ejemplos anteriores, pero pueden utilizarse siempre que un delegado requiera un método que pueda ser exhaustivo con una expresión única.

Utilizar una lambda con un delegado involucra dos pasos. Primero, debes declarar el tipo de delegado en sí. Segundo, cuando declaras una instancia del delegado, asignale la expresión lambda. Una vez que se ha realizado esto, la expresión lambda puede ser ejecutada invocando la instancia del delegado.

El siguiente ejemplo muestra el uso de una expresión lambda con un delegado. Declara dos tipos de delegados. Luego, asigna expresiones lambda a instancias de esos delegados. Finalmente, ejecuta las expresiones lambda a través de las instancias de delegado.

```
// Muestra el funcionamiento de las expresiones lambda.
using System;

// Primero, declara dos tipos de delegados.

// El delegado Transforma toma un argumento double
// y regresa un valor double.
delegate double Transforma(double v);

// El delegado TestInts toma dos argumentos int
// y regresa un resultado bool.
delegate bool TestInts(int w, int v);

class ExpresLambdaDemo {

    static void Main() {

        // Crea una expresión lambda que regresa el
        // recíproco de un valor.
        Transforma reciproco = n => 1.0 / n; ← Asigna una expresión
  lambda a un delegado.

        Console.WriteLine("El recíproco de 4 es " + reciproco(4.0));
        Console.WriteLine("El recíproco de 10 es " + reciproco(10.0));

        Console.WriteLine();

        // Crea una expresión lambda que determina si un
        // entero es factor de otro.
        TestInts esFactor = (n, d) => n % d == 0; ←
    }
}
```

```
        Console.WriteLine("¿Es 3 un factor de 9? " + esFactor (9, 3));
        Console.WriteLine("¿Es 3 un factor de 10? " + esFactor (10, 3));
    }
}
```

Los datos generados por el programa son:

```
El reciproco de 4 es 0.25
El reciproco de 10 es 0.1

¿Es 3 un factor de 9? True
¿Es 3 un factor de 10? False
```

En el programa, primero observa cómo se declaran los delegados. El delegado **Transforma** toma un argumento **double** y regresa un resultado **double**. El delegado **TestInts** toma dos argumentos **int** y regresa un resultado **bool**. A continuación, presta especial atención a estas declaraciones:

```
Transforma reciproco = n => 1.0 / n;
TestInts esFactor = (n, d) => n % d == 0;
```

La primera de ellas asigna una expresión lambda a **recíproco** que regresa el recíproco del valor que le es transmitido. Esta expresión puede ser asignada a un delegado **Transforma** porque es compatible con la declaración de **Transforma**. El argumento utilizado en la invocación al reciproco es transmitido a **n**. El valor regresado es el resultado de la expresión **1.0/n**.

La segunda declaración asigna a **esFactor** una expresión que regresa un valor true si el segundo argumento es un factor del primero. Esta lambda toma dos argumentos y regresa true si el primero puede ser dividido sin residuo entre el segundo. Así, es compatible con la declaración **TestInts**. Los dos argumentos transmitidos a **esFactor()** cuando es invocado son automáticamente transmitidos a **n** y **d**, en ese orden. Otro punto: los paréntesis alrededor de los parámetros **n** y **d** son necesarios. Los paréntesis son opcionales sólo cuando se utiliza un parámetro.

## Declaraciones lambda

Una declaración lambda expande los tipos de operaciones que pueden ser manejados directamente dentro de una expresión lambda. Por ejemplo, utilizando una declaración lambda, puedes hacer uso de loops, declaraciones condicionales **if**, declarar variables y demás. Una declaración lambda es fácil de crear. Simplemente encierra su cuerpo entre llaves.

He aquí un ejemplo que utiliza una declaración lambda para calcular y regresar el factorial de un valor **int**:

```
// Muestra el funcionamiento de una declaración lambda.
using System;

// IntOp toma un argumento int y regresa un resultado int.
delegate int IntOp(int end);

class DeclaraLambdaDemo {
```

```

static void Main() {
    // Una declaración lambda que regresa el factorial
    // del valor que le es transmitido.
    IntOp fact = n => {
        int r = 1;
        for(int i=1; i <= n; i++)
            r = i * r;
        return r;
    };
    Console.WriteLine("El factorial de 3 es " + fact (3));
    Console.WriteLine("El factorial de 5 es " + fact (5));
}

```

← Ésta es la declaración lambda.

Los datos generados por el programa son:

```

El factorial de 3 es 6
El factorial de 5 es 120

```

En el programa, observa que la declaración lambda declara una variable llamada **r**, utiliza un loop **for** y tiene una declaración **return**. Todos ellos son legales dentro de la declaración lambda. En esencia, una declaración lambda es un paralelo muy cercano a un método anónimo. Por ello, muchos métodos anónimos serán convertidos en declaraciones lambda cuando se actualice el código antiguo. Otro punto: cuando ocurre una declaración **return** dentro de una expresión lambda, simplemente provoca un regreso desde lambda. No provoca que regrese el método encerrado.

## Prueba esto Utilizar expresiones lambda para implementar controladores de eventos

Como las expresiones lambda pueden asignarse a delegados, también pueden ser utilizadas como controladores de eventos. En esta capacidad, las expresiones lambda pueden utilizarse en lugar de los métodos anónimos en muchos casos. El ejemplo desarrollado a continuación ilustra el uso de una expresión y una declaración lambda como controladores de eventos.

## Paso a paso

1. Crea un delegado de evento llamado **MiControladorEventos** y un evento llamado **MiEvento**, como se muestra a continuación:

```

// Declara un tipo delegado para un evento.
delegate void MiControladorEventos();

// Declara una clase con un evento.
class MiEvento {

```

(continúa)

```
public event MiControladorEventos UnEvento;

// Esto se invoca para disparar el evento.
public void Dispara() {
    if(UnEvento != null)
        UnEvento();
}
```

- 2.** Comienza a crear una clase llamada **ControladoresEventosLambda** que generará y controlará eventos, como se muestra aquí:

```
class ControladoresEventosLambda {
    static void Main() {
        MiEvento evt = new MiEvento();
        int cuenta = 0;
```

Observa que **Main()** crea un evento que es referido por **evt** y declara **cuenta** como una variable de enteros que se inicializa con el valor cero.

- 3.** Añade un controlador de eventos que incremente **cuenta** a la cadena de eventos **evt**, como se muestra a continuación:

```
// Esta expresión lambda incrementa el valor de cuenta cuando
// ocurre el evento.
evt.UnEvento +=() => cuenta++;
```

Este código utiliza una expresión lambda como controlador de eventos. Observa que puede utilizar la variable externa **cuenta**. Las mismas reglas respecto al uso de variables externas que se aplican a los métodos anónimos (descritas en el capítulo 12) también se aplican a las expresiones lambda.

- 4.** Añade un controlador de eventos a **evt** que muestra el valor de **cuenta**, como se muestra a continuación:

```
// Esta declaración lambda muestra el valor de cuenta.
// Si cuenta es mayor que 3, se reajusta a 0.
evt.UnEvento +=() => {
    if(cuenta > 3) cuenta = 0;
    Console.WriteLine("Cuenta es " + cuenta);
};
```

Esta porción de código utiliza una declaración lambda para mostrar el valor actual de **cuenta**. Antes de hacerlo, verifica el valor de **cuenta**. Si es mayor que 3, se reajusta a cero.

- 5.** Completa **ControladoresEventosLambda** invocando el evento cinco veces:

```
// Dispara el evento cinco veces.
evt.Dispara();
evt.Dispara();
evt.Dispara();
evt.Dispara();
evt.Dispara();
}
}
```

**6.** He aquí el ejemplo completo:

```
// Utiliza expresiones lambda como controladores de eventos.  
using System;  
  
// Declara un tipo delegado para un evento.  
delegate void MiControladorEventos();  
  
// Declara una clase con un evento.  
class MiEvento {  
    public event MiControladorEventos UnEvento;  
  
    // Esto se invoca para disparar el evento.  
    public void Dispara() {  
        if(UnEvento != null)  
            UnEvento();  
    }  
}  
  
class ControladoresEventosLambda {  
    static void Main() {  
        MiEvento evt = new MiEvento();  
        int cuenta = 0;  
  
        // Utiliza expresiones lambda para definir controladores de  
        // eventos.  
        // Esta expresión lambda incrementa el valor de cuenta cuando  
        // ocurre el evento.  
        evt.UnEvento +=() => cuenta++;  
  
        // Esta declaración lambda muestra el valor de cuenta.  
        // Si cuenta es mayor que 3, se reajusta a 0.  
        evt.UnEvento +=() => {  
            if(cuenta > 3) cuenta = 0;  
            Console.WriteLine("Cuenta es " + cuenta);  
        };  
  
        // Dispara el evento cinco veces.  
        evt.Dispara();  
        evt.Dispara();  
        evt.Dispara();  
        evt.Dispara();  
        evt.Dispara();  
    }  
}
```

(continúa)

Los datos generados por el programa son:

```
Cuenta es 1  
Cuenta es 2  
Cuenta es 3  
Cuenta es 0  
Cuenta es 1
```

## Pregunta al experto

**P:** OK, ¡estoy convencido! LINQ es increíblemente poderoso. ¿Cuál es la mejor manera de comenzar a aprender su uso?

**R:** Comienza explorando el contenido de **System.Linq**. Pon especial atención a las capacidades de los métodos de extensión definidos por **Enumerable**. A continuación, aumenta tu conocimiento y experiencia escribiendo expresiones lambda. Se espera que tengan un papel cada vez más importante en la programación de C#. Además, estudia las colecciones en **System.Collections** y **System.Collections.Generic**. En el capítulo 15 se presenta una introducción a las colecciones, pero hay mucho más que aprender. Aunque muy nuevo, LINQ es ya una parte importante de C#, y se espera que su uso se incremente con el tiempo. En términos sencillos, LINQ será parte del futuro de todo programador de C#. El esfuerzo que hagas hoy será compensado con creces.

## ✓ Autoexamen Capítulo 14

- 1.** ¿Qué significa LINQ? En términos generales, ¿cuál es su propósito?
- 2.** Cuando se relaciona con LINQ, ¿qué interfaz debe implementar una fuente de datos?
- 3.** ¿Cuáles son las palabras clave de query con las que principian sus cláusulas?
- 4.** Suponiendo que existe una fuente de datos llamada **miListaDatos**, muestra cómo crear una consulta que regrese una secuencia de esos objetos. Llama a la variable de query **todoData**.
- 5.** Suponiendo que **miListaDatos** contiene objetos **MiData** y que estos últimos definen una propiedad llamada **Altura**, reescribe la consulta de la pregunta 4 para que regrese una secuencia de la propiedad **Altura**, en lugar del objeto entero **MiData**.
- 6.** ¿Qué palabra clave de query se utiliza para filtrar una secuencia? Utilízala para reescribir tu respuesta a la pregunta 5, para que regrese sólo aquellos objetos **MiData** cuya propiedad **Altura** sea menor que 100.
- 7.** ¿Qué palabra clave de query ordena una secuencia? Utilizando tu respuesta para la pregunta 6, ordena el resultado en orden descendente con base en la propiedad **Altura**.

- 8.** ¿Qué palabra clave de query agrupa el resultado de una consulta organizándola en secuencias? (En otras palabras, ¿qué palabra clave regresa una secuencia de secuencias?) Muestra su formato general.
- 9.** ¿Qué palabra clave une dos secuencias? Muestra su formato general en el contexto de una cláusula **from**.
- 10.** Cuando utilizas **select** o **group**, ¿qué palabra clave crea una continuidad?
- 11.** ¿Qué palabra clave crea una variable que aloja un valor?
- 12.** Muestra cómo crear una instancia de un tipo anónimo que tenga dos propiedades **string** llamadas **Título** y **Autor**.
- 13.** ¿Qué es un operador lambda?
- 14.** Una expresión lambda es una forma de función anónima. ¿Cíerto o falso?
- 15.** ¿Cuál es el método de consulta que corresponde a la palabra clave **where**?
- 16.** ¿Qué es un método de extensión? ¿Cómo debe ser declarado su primer parámetro?
- 17.** ¿Por qué son importantes para LINQ los métodos de extensión?



# Capítulo 15

El preprocesador, RTTI,  
tipos anulables y otros  
temas avanzados

## Habilidades y conceptos clave

- Preprocesador
  - Identificación de tipo en tiempo de ejecución
  - Tipos anulables
  - Código inseguro
  - Atributos
  - Operadores de conversión
  - Colecciones
  - Palabras clave adicionales
- 

**H**as recorrido un largo camino desde el inicio de este libro. Este capítulo final examina varios temas C#, como el preprocesador, tipo ID de tiempo de ejecución y tipos anulables, que no caben fácilmente en ninguno de los capítulos anteriores. También presenta una introducción a las colecciones, hace una descripción general de código inseguro y muestra cómo crear un operador de conversiones. El capítulo concluye con una breve mirada a las restantes palabras clave de C# que no se cubrieron en los capítulos anteriores. Muchos de los temas de este capítulo se aplican a usos avanzados de C# y la explicación en detalle de los mismos rebasa los alcances de este libro. Se presentan para que tengas una imagen completa de las posibilidades de C#.

## El preprocesador

C# define varias *directivas de preprocesador*, que afectan la manera como el compilador interpreta el código fuente de tu programa. Estas directivas afectan el texto de tu archivo fuente en donde ocurren, antes de la traducción del programa a código objeto. El término *directivas de preprocesador* viene del hecho de que estas instrucciones fueron manejadas tradicionalmente por una fase de compilación llamada *preprocesamiento*. Hoy en día la tecnología de los procesadores modernos no requiere una etapa de procesamiento previo para manejar las directivas, pero el nombre ha permanecido.

C# define las siguientes directivas de preprocesamiento:

|            |         |        |          |
|------------|---------|--------|----------|
| #define    | #elif   | #else  | #endif   |
| #endregion | #error  | #if    | #line    |
| #pragma    | #region | #undef | #warning |

Todas las directivas de preprocesador inician con un signo #. Además, cada directiva de preprocesamiento debe ir en su propia línea.

Dada la arquitectura moderna y orientada a objetos de C#, no hay mucha necesidad de las directivas de preprocesador como la había en los lenguajes antiguos. Sin embargo, pueden ser de utilidad de cuando en cuando, en especial para la compilación condicional. A continuación se examina cada una de las directivas.

## #define

La directiva **#define** especifica una secuencia de caracteres que invocan un *símbolo*. La existencia o no existencia de un símbolo puede ser determinada por **#if** o **#elif**, y es utilizada para controlar la compilación.

`#define símbolo`

Observa que esta declaración no finaliza con punto y coma. Puede haber cualquier cantidad de espacios entre **#define** y el símbolo, pero una vez que este último inicia, su final sólo es marcado por un carácter de nueva línea. Por ejemplo, para definir el símbolo **EXPERIMENTAL**, se utiliza la directiva:

```
#define EXPERIMENTAL
```

## #if y #endif

Las directivas **#if** y **#endif** te permiten compilar condicionalmente una secuencia de código dependiendo de si una expresión que involucra uno o más símbolos es evaluada como verdadera. Un símbolo es verdadero (true) si ha sido definido. De lo contrario es falso (false). De esta manera, si un símbolo ha sido definido por la directiva **#define**, será evaluado como verdadero.

El formato general de **#if** es

```
#if símbolo-expresión
    secuencia de declaraciones
#endif
```

Si la expresión posterior a **#if** es verdadera, el código que se encuentra entre ella y **#endif** se compila. En caso contrario, el compilador lo salta. La directiva **#endif** marca el final del bloque **#if**.

Una expresión de símbolo puede ser tan sencilla como el nombre del símbolo. También puedes utilizar operadores en una expresión de símbolo: **!**, **==**, **!=**, **&&** y **||**. También se permite el uso de paréntesis.

A continuación presentamos un ejemplo que muestra el uso de la compilación condicional:

```
// Muestra el uso de #if, #endif y #define.
#define EXPERIMENTAL

using System;
class Test {
    static void Main() {
        #if EXPERIMENTAL
            Console.WriteLine("Compilado para la versión experimental.");
        #endif
    }
}
```

Compilado sólo si se define EXPERIMENTAL.

```
        Console.WriteLine("Esto está presente en todas las versiones.");
    }
}
```

El programa presenta los siguientes datos de salida:

```
Compilado para la versión experimental.  
Esto está presente en todas las versiones.
```

El programa define el símbolo **EXPERIMENTAL**. De esta manera, cuando **#if** es localizado por el compilador, la expresión del símbolo se evalúa como verdadera, y la primera declaración **WriteLine()** se compila. Si eliminas la descripción de **EXPERIMENTAL** y recompilas el programa, la primera declaración **WriteLine()** no se compilará porque el **#if** será evaluado como falso. En todos los casos la segunda declaración **WriteLine()** será compilada porque no forma parte del bloque **#if**.

Como ya se explicó, puedes utilizar una expresión de símbolo en un **#if**. Por ejemplo:

```
// Utiliza una expresión de símbolo.  
#define EXPERIMENTAL  
#define ENSAYO  
  
using System;  
  
class Test {  
    static void Main() {  
  
        #if EXPERIMENTAL  
            Console.WriteLine("Compilado para la versión experimental.");  
        #endif  
  
        #if EXPERIMENTAL && ENSAYO ← Una expresión de símbolo.  
            Console.Error.WriteLine("Prueba versión experimental de ensayo.");  
        #endif  
  
        Console.WriteLine("Esto está presente en todas las versiones.");  
    }  
}
```

Los datos generados por este programa son:

```
Compilado para la versión experimental.  
Prueba versión experimental de ensayo.  
Esto está presente en todas las versiones.
```

En este ejemplo se definen dos símbolos, **EXPERIMENTAL** y **ENSAZO**. La segunda declaración **WriteLine()** se compila sólo si ambos son definidos.

Puedes utilizar el **!** para compilar código cuando un símbolo no está definido. Por ejemplo,

```
#if !CC_PASA  
    Console.WriteLine("El código no ha pasado el control de calidad.")  
#endif
```

La invocación a **WriteLine()** será compilada sólo si **CC\_PASA** no ha sido definido.

## #else y #elif

La directiva **#else** funciona de manera muy parecida al **else** que forma parte del lenguaje C#: establece una alternativa en caso de que **#if** falle. He aquí un ejemplo:

```
// Muestra el funcionamiento de #else.

#define EXPERIMENTAL

using System;

class Test {
    static void Main() {

        #if EXPERIMENTAL
            Console.WriteLine("Compilado para la versión experimental.");
        #else ←
            Console.WriteLine("Compilado para la liberación.");
        #endif

        #if EXPERIMENTAL && TRIAL
            Console.Error.WriteLine("Probando versión experimental de
            ensayo.");
        #else ←
            Console.Error.WriteLine("Versión de ensayo no experimental.");
        #endif

        Console.WriteLine("Esto está presente en todas las versiones.");
    }
}
```

Usa **#else**.

Los datos generados son los siguientes:

```
Compilado para la versión experimental.
Versión de ensayo no experimental.
Esto está presente en todas las versiones.
```

Como **ENSAYO** no está definido, se utiliza la porción **#else** del segundo código condicional.

Observa que **#else** marca tanto el final del bloque **#if** como el principio del bloque **#else**. Esto es necesario porque sólo puede haber un **#endif** asociado con cualquier **#if**. Más aún, sólo puede haber un **#else** asociado con cualquier **#if**.

La directiva **#elif** significa “else if” y establece una cadena if-else-if para múltiples opciones de compilación. **#elif** es seguida por una expresión de símbolo. Si la expresión es verdadera, se compila ese bloque de código y no se prueba ninguna otra expresión **#elif**. De lo contrario, se verifica el siguiente bloque de la serie. Si ningún **#elif** tiene éxito, y si existe un **#else**, la secuencia de código asociada con este último se compila. En caso contrario, no se compila ningún código dentro de la directiva **#if**.

## 558 Fundamentos de C# 3.0

---

He aquí un ejemplo para mostrar el funcionamiento de **#elif**:

```
// Muestra el funcionamiento de #elif.
#define RELEASE

using System;

class Test {
    static void Main() {

        #if EXPERIMENTAL
            Console.WriteLine("Compilado para la versión experimental.");
        #elif RELEASE ←———— Usa #elif.
            Console.WriteLine("Compilado para la liberación.");
        #else
            Console.WriteLine("Compilado para prueba interna.");
        #endif

        #if TRIAL && !RELEASE
            Console.WriteLine("Versión de ensayo.");
        #endif

        Console.WriteLine("Esto está presente en todas las versiones.");
    }
}
```

Los datos generados son:

```
Compilado para la liberación.
Esto está presente en todas las versiones.
```

Uniendo todas las piezas, a continuación presentamos el formato general de las directivas **#if/#else/#elif/#endif**:

```
#if expresión-símbolo
    secuencia de declaraciones
#elif expresión-símbolo
    secuencia de declaraciones
#elif expresión-símbolo
    secuencia de declaraciones
    .
    .
    .
#else expresión-símbolo
    secuencia de declaraciones
#endif
```

## #undef

La directiva **#undef** elimina un símbolo definido anteriormente. Es decir, “indefine” un símbolo. El formato general de **#undef** es:

**#undef** *símbolo*

Por ejemplo:

```
#define DISPOSITIVO_MOVIL
#if DISPOSITIVO_MOVIL
// ...
#undef DISPOSITIVO_MOVIL
// En este punto DISPOSITIVO_MOVIL queda sin definición.
```

Después de la directiva **#undef**, **DISPOSITIVO\_MOVIL** ya no está definido.

**#undef** se utiliza principalmente para limitar el uso de los símbolos a las secciones del código donde son necesarios.

## Pregunta al experto

**P:** He notado que las directivas de preprocesador de C# tienen muchas similitudes con las directivas de preprocesador soportadas por C y C++. Más aún, en C/C++, sé que puedes usar **#define** para realizar sustituciones textuales, como definir un nombre para un valor y para crear funciones parecidas a las macros. ¿C# soporta tales usos para **#define**?

**R:** No. En C#, **#define** se utiliza sólo para definir un símbolo.

## #error

La directiva **#error** fuerza al compilador a detener la compilación. Se utiliza para la depuración. El formato general de **#error** es:

**#error** *mensaje-error*

Cuando se encuentra la directiva **#error**, se muestra el mensaje de error. Por ejemplo, cuando el compilador encuentra una línea como la siguiente:

```
#error ¡El código depurado aún está siendo compilado!
```

la compilación se detiene y se muestra el mensaje de error “¡El código depurado aún está siendo compilado!”

## #warning

La directiva **#warning** es parecida a **#error**, sólo que se produce una advertencia en lugar de un error. Así, la compilación no se detiene. El formato general de la directiva **#warning** es:

**#warning** *mensaje-advertencia*

## #line

La directiva **#line** establece el número de línea y el nombre de archivo para el archivo que contiene esta directiva. Dichos número y nombre se utilizan cuando surgen errores o advertencias durante la compilación. El formato general de **#line** es:

```
#line número “nombre de archivo”
```

donde *número* es cualquier entero positivo y se convierte en el número de la nueva línea, y la parte opcional *nombre de archivo* es cualquier identificador de archivo válido, que se convierte en el nuevo nombre de archivo.

**#line** permite dos opciones. La primera de ellas es **default**, que regresa el número de línea a su condición original. Se usa así:

```
#line default
```

La segunda es **hidden**. Cuando recorre el programa, la opción **hidden** permite que un depurador pase de largo líneas entre una directiva

```
#line hidden
```

y la siguiente directiva **#line** que no incluye la opción **hidden**.

## #region y #endregion

Las directivas **#region** y **#endregion** te permiten definir una región que será expandida o contraída por el IDE de Visual Studio cuando se utiliza la característica de esbozo. El formato general se muestra a continuación:

```
#region  
  // secuencia de código  
#endregion
```

## #pragma

La directiva **#pragma** le da instrucciones al compilador, como especificar una opción. Tiene el siguiente formato general:

```
#pragma opción
```

Aquí, *opción* es la instrucción que se transmite al compilador.

En C# 3.0 hay dos opciones soportadas por **#pragma**. La primera es **warning**, que se utiliza para activar o desactivar advertencias específicas del compilador. Tiene los siguientes dos formatos:

```
#pragma warning disable advertencias
```

```
#pragma warning restore advertencias
```

Aquí, *advertencias* es una lista de números de advertencia separados por comas. Para desactivar una advertencia utiliza la opción **disable**. Para activar una advertencia, utiliza la opción **restore**.

Por ejemplo, la siguiente declaración **#pragma** desactiva la advertencia 168, que indica cuan-  
do una variable es declarada pero no se utiliza:

```
#pragma warning disable 168
```

La segunda opción de **#pragma** es **checksum**. Se usa para generar una suma de verificación  
para proyectos ASP.NET. Tiene el siguiente formato general:

```
#pragma checksum "nombre de archivo" "{GUID}" "check-sum"
```

Aquí, *nombre de archivo* es, precisamente, el nombre del archivo, *GUID* es el identificador único  
global asociado con el nombre de archivo y *check-sum* es un número hexadecimal que contiene la  
suma de verificación. La cadena debe contener un número par de dígitos.

## Identificación del tipo tiempo de ejecución

En C# es posible determinar el tipo de un objeto en tiempo de ejecución. De hecho, C# incluye  
tres palabras clave que soportan la identificación de tipo en tiempo de ejecución: **is**, **as** y **typeof**.  
Como la identificación de tipos durante la ejecución es una de las características más avanzadas de  
C#, es importante tener un conocimiento general sobre el tema.

### Probar un tipo con **is**

Puedes determinar si un objeto pertenece a cierto tipo de dato utilizando el operador **is**. Su formato  
general se muestra a continuación:

*obj* **is** *tipo*

Aquí, *obj* es una expresión que describe un objeto cuyo tipo va a ser verificado contra *tipo*. Si el  
tipo de *obj* es el mismo o compatible con *tipo*, entonces, el resultado de la operación es verdadero.  
De lo contrario, es falso. Así, si el resultado es verdadero, *obj* puede ser transformado a *tipo*. He  
aquí un ejemplo:

```
// Muestra el funcionamiento de is.
using System;

class A {}
class B : A {}

class UseIs {
    static void Main() {
        A a = new A();
        B b = new B();

        if(a is A) Console.WriteLine("a es una A");
        if(b is A) ← Esto es verdadero porque b es una A.
        Console.WriteLine("b es una A porque es una derivada de A");
```

```
if(a is B) ← Esto es falso porque a no es una B.  
Console.WriteLine("Esto no se mostrará -- porque a no deriva de B");  
  
if(b is B) Console.WriteLine("b es una B");  
if(a is object) Console.WriteLine("a es un Objeto");  
}  
}
```

Los datos generados por el programa son:

```
a es una A  
b es una A porque es una derivada de A  
b es una B  
a es un Objeto
```

La mayoría de las expresiones **is** se explican por sí solas, pero dos de ellas merecen una mirada más cercana. Primero, observa esta declaración:

```
if(b is A)  
Console.WriteLine("b es una A porque es una derivada de A");
```

el **if** se ejecuta con éxito porque **b** es un objeto de tipo **B**, que a su vez deriva del tipo **A**. De esta manera, **b** es compatible con **A**. Sin embargo, no sucede lo mismo a la inversa. Cuando se ejecuta esta línea:

```
if(a is B)  
Console.WriteLine("Esto no se mostrará -- porque a no deriva de B");
```

el **if** no se ejecuta porque **a** es de tipo **A**, que no deriva de **B**. Por ello, no son compatibles.

## Utilizar **as**

En algunas ocasiones querrás hacer conversiones durante la ejecución del programa sin lanzar una excepción en caso de que la conversión falle (que es el caso cuando se utiliza una transformación). Para hacer esto utiliza el operador **as**, que tiene este formato general:

*expr as tipo*

Aquí, *expr* es la expresión que será convertida a *tipo*. Si la conversión es exitosa, entonces se regresa una referencia a *tipo*. En caso contrario, regresa una referencia nula. El operador **as** sólo puede ser utilizado para realizar referencias, encajonar, desencajonar o identificar conversiones.

## Utilizar **typeof**

Puedes obtener información sobre un tipo dado utilizando **typeof**, que tiene el siguiente formato general:

**typeof** (*tipo*)

Aquí, *tipo* es el tipo que será obtenido. Regresa una instancia de **System.Type** que es una clase que presenta la información asociada con el tipo. Utilizando esta instancia, puedes recuperar

información sobre el tipo. Por ejemplo, este programa muestra el nombre completo de la clase **StreamReader**:

```
// Muestra el funcionamiento de typeof.  
using System;  
using System.IO;  
  
class UseTypeof {  
    static void Main() {  
        Type t = typeof(StreamReader);  
        Console.WriteLine(t.FullName);  
    }  
}
```

El programa muestra el siguiente resultado:

```
System.IO.StreamReader
```

**System.Type** contiene muchos métodos, campos y propiedades que describen un tipo. Con seguridad querrás explorarlo por cuenta propia.

## Tipos anulables

Desde la versión 2.0, C# ha incluido una característica que proporciona una elegante solución a un problema que es tan común como irritante. La característica son *los tipos anulables*. El problema es cómo reconocer y manejar campos que no tienen ningún valor (en otras palabras, campos sin asignación). Para comprender el problema, considera una sencilla base de datos de clientes que mantiene los registros de nombre, dirección, ID, número de factura y balance actual. En tal situación, es posible crear una entrada de cliente en la cual uno o más de los campos no tengan asignación. Por ejemplo, es posible que el cliente simplemente haya solicitado un catálogo. En ese caso no se requerirá un número de factura y el campo no será utilizado.

Antes de los tipos anulables, manejar la posibilidad de campos superfluos requería ya fuera el uso de valores marcadores de posición o un campo extra que simplemente indicara si el campo estaba en uso o no. Por supuesto, los valores para marcar posición funcionaban sólo si existía un valor que de otra manera no fuera válido, lo cual no sucedía en todas las ocasiones. Añadir un campo extra para indicar si un campo estaba en uso o no funcionaba en todas las ocasiones, pero crear y administrar manualmente un campo así es una molestia. Los tipos anulables resuelven ambos problemas.

Un tipo anulable es una versión especial de un tipo valor que está representado por una estructura. Además de los valores definidos por el tipo subyacente, un tipo anulable puede almacenar el valor **null**. De esta manera, un valor anulable tiene el mismo rango y características que su tipo subyacente. Simplemente añade la capacidad de representar un valor que indica que la variable de ese tipo no tiene asignación. Los tipos anulables son objetos de **System.Nullable<T>**, donde **T** debe ser un tipo valor no anulable.

Un tipo anulable puede ser especificado de dos maneras diferentes. La primera, puedes utilizar explícitamente el tipo **Nullable<T>**. Por ejemplo, el siguiente código declara variables de tipos anulables **int** y **bool**:

```
Nullable<int> cuenta;  
Nullable<bool> hecho;
```

La segunda manera de especificar un tipo anulable es más corta y de mayor uso. Simplemente coloca un ? después del nombre del tipo subyacente. Por ejemplo, el siguiente código muestra la manera más común de declarar variables de tipos anulables **int** y **bool**:

```
int? cuenta;  
bool? hecho;
```

Cuando se utilizan tipos anulables, en muchas ocasiones encontrarás un objeto anulable creado de la siguiente manera:

```
int? cuenta = null;
```

Esto inicializa explícitamente **cuenta** como **null**. Esto satisface la restricción de que una variable debe tener cierto valor antes de ser utilizada. En este caso, el valor simplemente significa indefinido.

Puedes asignar un valor a una variable anulable de la manera tradicional porque la conversión de un tipo subyacente a un tipo anulable está predefinida. Por ejemplo, el siguiente código asigna el valor 100 a **cuenta**:

```
cuenta = 100;
```

Existen dos maneras para determinar si una variable de tipo anulable es **null** o si contiene un valor. La primera consiste en comparar su valor contra **null**. Por ejemplo, utilizando la variable **cuenta** declarada en el ejemplo anterior, la siguiente línea determina si tiene un valor:

```
if(cuenta != null) // contiene un valor
```

Si **cuenta** no es **null**, entonces contiene un valor.

La segunda manera para determinar si una variable nula contiene un valor es utilizar la propiedad sólo-lectura **HasValue**, definida por **Nullable<T>**, como se muestra a continuación:

```
bool HasValue
```

**HasValue** regresará verdadero si la instancia sobre la cual se está invocando contiene un valor. En caso contrario, regresará falso. A continuación presentamos la segunda manera para determinar si el objeto anulable **cuenta** tiene un valor, utilizando la propiedad **HasValue**:

```
if(cuenta.HasValue) // contiene un valor
```

Suponiendo que un objeto anulable tiene un valor, puedes obtenerlo utilizando la propiedad sólo-lectura **Value**, definida por **Nullable<T>**, como se muestra a continuación:

```
T Value
```

La cual regresa el valor de la instancia anulable sobre la cual es invocada. Si intentas obtener un valor de una variable que es **null**, será lanzada una excepción **System.InvalidOperationException**. También es posible obtener el valor de una instancia anulable transformándola a su tipo subyacente.

El siguiente programa arma las piezas mencionadas y muestra el mecanismo básico para manejar un tipo anulable:

```
// Muestra un tipo anulable en acción.

using System;

class AnizableDemo {
    static void Main() {
        int? cuenta = null; ← Declara un tipo anizable para int.

        if(cuenta.HasValue) ←
            Console.WriteLine("cuenta tiene este valor: " + cuenta.Value);
        else
            Console.WriteLine("cuenta no tiene valor");

        cuenta = 100;

        if(cuenta.HasValue) ←
            Console.WriteLine("cuenta tiene este valor: " + cuenta.Value);
        else
            Console.WriteLine("cuenta no tiene valor");
    }
}
```

Utiliza **HasValue** para determinar si **cuenta** tiene un valor.

Los resultados que genera el programa son:

```
cuenta no tiene valor
cuenta tiene este valor: 100
```

## El operador ??

Si intentas utilizar el mecanismo de transformación para convertir un tipo anizable a su tipo subyacente, se lanzará una excepción **System.InvalidOperationException** si el tipo anizable contiene un valor **null**. Esto puede ocurrir, por ejemplo, cuando utilizas cast para asignar el valor de un objeto anizable a una variable de su tipo subyacente. Puedes evitar la posibilidad de que ocurra esta excepción utilizando el operador **??**, que recibe el nombre de *operador null coalescente*. Te permite especificar un valor por defecto que será utilizado cuando el objeto anizable contenga un valor **null**. También elimina la necesidad de la transformación.

El operador **??** tiene el siguiente formato general:

*objeto-anizable ?? valor-por-defecto*

Si *objeto-anizable* contiene un valor, entonces el valor de **??** es el mismo. En caso contrario, el valor de la operación **??** es *valor-por-defecto*.

Por ejemplo, en el siguiente código **balance** es **null**. Esto hace que a **balanceActual** se le asigne el valor 0.0 y no se lanzará ninguna excepción.

```
double ? balance = null;
double balanceActual;

balanceActual = balance ?? 0.0;
```

En la siguiente secuencia a **balance** se le da el valor de 123.75.

```
double ? balance = 123.75;
double balanceActual;

balanceActual = balance ?? 0.0;
```

Ahora, **balanceActual** contendrá el valor de **balance**, que es 123.75.

Otro punto: la expresión a la derecha de **??** es evaluada sólo si la expresión de la izquierda no contiene un valor.

## Objetos anulables y los operadores relacionales y lógicos

Los objetos anulables pueden ser utilizados en expresiones relacionales exactamente de la misma manera que con sus correspondientes tipos no anulables. Pero existe una regla adicional que se debe aplicar. Cuando se comparan dos objetos anulables utilizando los operadores **<**, **>**, **<=** o **=>**, el resultado es falso si cualesquiera de los operadores es **null**. Por ejemplo, considera esta secuencia:

```
byte ? bajo = 16;
byte ? alto = null;

// Aquí bajo está definido, pero alto no.
if(bajo > alto) // falso
```

Aquí, el resultado de la prueba para menor qué es falso. Sin embargo, de forma contraria a lo que se esperaría, también lo es en la comparación inversa:

```
if(bajo > alto) // .. también es falso!
```

Así, cuando uno (o ambos) de los objetos anulables utilizados en una comparación es **null**, el resultado de esa operación será siempre falso. De esta manera, **null** no participa en una relación ordenada.

No obstante, puedes verificar si un objeto anulable contiene **null** utilizando el operador **==** o **!=**. Por ejemplo, la siguiente es una operación válida que dará como resultado un valor verdadero:

```
if(alto == null) // ...
```

Cuando una expresión lógica involucra dos objetos **bool?**, el resultado de la expresión será uno de tres valores: **true**, **false** o **null** (sin definición). A continuación presentamos las entradas que son añadidas a la tabla de verdad para los operadores **&** y **|** que aplican a **bool?**:

| P     | Q     | P   Q | P & Q |
|-------|-------|-------|-------|
| true  | null  | true  | null  |
| false | null  | null  | false |
| null  | true  | true  | null  |
| null  | false | null  | false |

Otro punto: Cuando el operador **!** se aplica a un valor **bool?** que es **null**, el resultado es **null**.

## Código inseguro

C# te permite escribir lo que es conocido como ¡código “inseguro”! Esto puede sonar a código que contiene errores, pero no es así. El código inseguro no se refiere al código pobemente escrito. Es código que no se ejecuta bajo la gestión total del Lenguaje Común de Tiempo de Ejecución (CLR). Como se explicó en el capítulo 1, C# se utiliza normalmente para escribir código controlado. Es posible, sin embargo, escribir código que no se ejecute bajo control total del CLR. Como se trata de código sin vigilancia, no está sujeto al mismo escrutinio ni a las restricciones que el código controlado. Se le llama “inseguro” porque es imposible verificar que no realice algún tipo de acción dañina. Así, el término *inseguro* no significa que el código esté necesariamente viciado. Significa que es posible que el código realice acciones que no están sujetas a la supervisión del contexto administrativo del sistema.

El código controlado, benéfico en la mayor parte, previene el uso de *punteros*. Si estás familiarizado con C o C++, entonces sabrás que los punteros son variables que contienen las direcciones de otros objetos. Así, conceptualmente, los punteros son un poco parecidos a las referencias de C#. La principal diferencia consiste en que un puntero puede apuntar hacia cualquier lugar de la memoria; una referencia siempre se refiere a un objeto de su tipo. Como los punteros pueden apuntar hacia cualquier lugar de la memoria, es posible que se haga un empleo erróneo de ellos. También resulta fácil introducir un error de código cuando se utilizan los punteros. Por ello C# no les da soporte cuando se crea código controlado. Los punteros son, sin embargo, tanto útiles como necesarios para cierto tipo de programación (como cuando se escribe código para interactuar con un dispositivo), y C# te permite crear y utilizar punteros. Todas las operaciones con punteros deben ser marcadas como inseguras, porque se ejecutan fuera del ambiente controlado.

Como punto de interés, las declaraciones y el uso de los punteros en C# son paralelos a los de C/C++; si sabes utilizar los punteros en C/C++, entonces sabes usarlos en C#. Pero recuerda que la intención de C# es utilizar código controlado. Su capacidad para dar soporte a código inseguro le permite aplicarse a tipos especiales de problemas. No está hecho para la programación normal de C#. De hecho, para compilar código inseguro, debes utilizar la opción de compilación **/unsafe**. En general, si necesitas crear grandes cantidades de código que se ejecuten fuera del CLR, entonces sería mejor que utilizaras C++.

Trabajar con código inseguro es un tema avanzado, y una explicación exhaustiva del tema rebasa los alcances de este libro. Dicho esto, examinaremos brevemente los punteros y dos palabras clave que dan soporte a código inseguro: **unsafe** y **fixed**.

### Una breve mirada a los punteros

Un puntero es una variable que contiene la dirección de algún otro objeto. Por ejemplo, si **p** contiene la dirección de **y**, entonces se dice que **p** “apunta hacia” **y**. Las variables de puntero deben declararse como tales. El formato general de una declaración de variable de puntero es

*tipo*\* *nombre-var*;

Aquí, *tipo* es el tipo de objeto hacia el cual señalará el puntero, y debe ser un tipo diferente al de referencia; *nombre-var* es el nombre de la variable de puntero. Por ejemplo, para declarar **ip** de manera que sea un puntero hacia **int**, utiliza esta declaración:

```
int* ip;
```

Para un puntero **float** utiliza

```
float* fp;
```

Como regla general, un tipo puntero se crea en cualquier declaración donde el nombre del tipo va seguido por un **\***.

El tipo de dato al que se dirigirá un puntero está determinado por su *tipo referente*, que es comúnmente conocido como el tipo de base del puntero. De esta manera, en los ejemplos anteriores, **ip** puede ser utilizado para apuntar hacia un **int**, y **fp** puede utilizarse para apuntar a un **float**. Sin embargo, debes comprender que no hay nada que impida a un apuntador señalar hacia cualquier otro lado. Por ello los apuntadores son potencialmente inseguros.

Recuerda que un tipo puntero puede ser declarado sólo para tipos que no sean referencia. Esto significa que el tipo referente de un puntero puede ser cualquiera de los tipos simples, como **int**, **double** y **char**; un tipo de enumeración; o un **struct** (siempre y cuando ninguno de sus campos sean tipos de referencia).

Existen dos operadores de puntero claves: **\*** y **&**. La **&** es un operador unitario que regresa la dirección de memoria de su operando. (Recuerda que un operador unitario es aquel que sólo requiere un operando.) Por ejemplo:

```
int* ip;
int num = 10;

ip = &num;
```

coloca en **ip** la dirección en memoria de la variable **num**. Esta dirección es la localización de la variable en la memoria interna de la computadora. *No tiene relación alguna con el valor de num*. Así, **ip** *no contiene* el valor 10 (valor inicial de **num**). Contiene la dirección en la cual está almacenado **num**. La operación de **&** puede ser recordada como “regresar la dirección de” la variable que le antecede. Por lo mismo, la declaración de arriba puede verbalizarse como “**ip** recibe la dirección de **num**”.

El segundo operador es **\***, y es el complemento de **&**. Se trata de un operador unitario que *condesciende* el puntero. En otras palabras, hace una evaluación a la variable localizada en la dirección especificada por su operando. Continuando con el mismo ejemplo, si **ip** contiene la dirección de memoria de la variable **num**, entonces

```
int val;
val = *ip;
```

colocará en **val** el valor 10, que es el valor de **num** (al que apunta **ip**). La operación de **\*** puede ser recordada por “en la dirección”. En este caso, entonces, la declaración puede ser leída como “**val** recibe el valor de la dirección **ip**”.

Los punteros también pueden ser utilizados dentro de estructuras. Cuando accedes el miembro de una estructura a través de un puntero, debes utilizar el operador **->** en lugar del operador de punto **(.)**. El operador **->** es llamado informalmente *operador de flecha*. Por ejemplo, dada la estructura:

```
struct MiEstructura {
    public int x;
    public int y;
    public int sum() { return x + y; }
}
```

he aquí cómo accesarías sus miembros a través de un puntero:

```
MiEstructura o = new MiEstructura();
MiEstructura* p; // declara un puntero

p = &o;
p->x = 10;
p->y = 20;

Console.WriteLine("Suma es " + p->sum());
```

Los punteros pueden tener simples operaciones aritméticas que se ejecutan en ellos. Por ejemplo, puedes incrementar o reducir un puntero. Al hacerlo se provoca que apunte al siguiente o posterior objeto de su tipo referente. También puedes sumar o restar valores enteros desde o hacia un puntero. Puedes restar un puntero de otro (el cual produce la cantidad de elementos del tipo referente que separa a los dos), pero no puedes sumar punteros.

## Pregunta al experto

**P:** Sé que cuando se declara un puntero en C++, el \* no es distributivo sobre una lista de variables en una declaración. Así, en C++, la declaración

```
int* p, q;
```

declara un puntero de entero llamado p y un entero llamado q. Es equivalente a las siguientes dos declaraciones:

```
int*p;
int q;
```

¿Es lo mismo en C#?

**R:** No. En C#, el \* *es* distributivo y la declaración

```
int* p, q;
```

crea dos variables de puntero. Así, es lo mismo que las siguientes dos declaraciones:

```
int* p;
int* q;
```

Es una diferencia importante que se debe considerar cuando se traslada código de C++ a C#.

## La palabra clave unsafe

Cualquier código que utiliza punteros debe ser marcado como inseguro por la palabra clave **unsafe**. Puedes marcar tipos (como clases o estructuras), miembros (como métodos y operadores), o bloques individuales de código inseguro. Por ejemplo, a continuación presentamos un programa que utiliza punteros dentro de **Main()**, y que está marcado como inseguro:

```
// Muestra el uso de punteros y unsafe.
// Necesitas compilar este programa utilizando la opción /unsafe.
using System;

class CodInseguro {
    // Marca Main como inseguro.
    unsafe static void Main() { ← Main() está marcado como inseguro
        int cuenta = 99;
        int* p; // crea un puntero int

        p = &cuenta; // coloca la dirección de cuenta en p
        Console.WriteLine("El valor inicial de cuenta es " + *p);

        *p = 10; // asigna a cuenta a través de p
        Console.WriteLine("Nuevo valor de cuenta es " + *p);
    }
}
```

Los datos generados por el programa son:

```
El valor inicial de cuenta es 99
Nuevo valor de cuenta es 10
```

Este programa utiliza el puntero **p** para obtener el valor contenido en **cuenta**, que es el objeto al cual está dirigido **p**. A causa de las operaciones de puntero, debe ser marcado como inseguro para que sea compilado.

## Utilizar fixed

La palabra clave **fixed** (fijo) tiene dos usos. El primero es prevenir que un objeto controlado sea recogido por el recolector de basura. Esto es necesario cuando un puntero hace referencia a un campo dentro de dicho objeto, por ejemplo. Como el puntero no tiene conocimiento de las acciones del recolector de basura, si el objeto se mueve, el puntero señalará hacia la localización equivocada.

He aquí el formato general de **fixed**:

```
fixed (tipo* p=&varFija) {
    // usa objeto fijo
}
```

Aquí, *p* es un puntero al que se le ha asignado la dirección de una variable. La variable permanecerá en su localización actual dentro de la memoria hasta que el bloque de código haya sido

ejecutado. También puedes utilizar una sola declaración como blanco de una declaración **fixed**. La palabra clave **fixed** puede utilizarse sólo en un contexto inseguro.

He aquí un ejemplo de **fixed**:

```
// Muestra el uso de fixed.
using System;

class Test {
    public int num;
    public Test(int i) { num = i; }
}

class UsaFixed {
    // Marca Main como inseguro.
    unsafe static void Main() {
        Test o = new Test(19);
        Utiliza fixed para fijar la localización de o.
        ↓
        fixed (int* p = &o.num) { // usa fixed para colocar la dirección de
            o.num en p
            Console.WriteLine("El valor inicial de o.num es " + *p);

            *p = 10; // asigna a o.num a través de p
            Console.WriteLine("Nuevo valor de o.num es " + *p);
        }
    }
}
```

Los datos generados por el programa son:

```
El valor inicial de o.num es 19
Nuevo valor de o.num es 10
```

En este caso, **fixed** evita que **o** sea movido. Esto es necesario porque **p** apunta hacia **o.num**. Si **o** se mueve, entonces **o.num** también se moverá. Esto provocaría que **p** apuntara hacia una locación inválida. El uso de **fixed** lo evita.

El segundo uso de **fixed** es crear arreglos unidimensionales de tamaño fijo. A esto se le llama *buffer de tamaño fijo*. Un buffer de tamaño fijo es siempre un miembro de **struct**. El propósito de un buffer de tamaño fijo es permitir la creación de **struct** en el cual los elementos del arreglo que conforman el buffer estén contenidos dentro de **struct**. Normalmente, cuando incluyes un miembro del arreglo en **struct**, dentro de éste sólo se conserva una referencia al arreglo. Esto da como resultado una estructura que puede utilizarse en situaciones dentro de las cuales el tamaño de **struct** es importante, como en la programación de lenguajes mezclados, interfaces de datos que no son creados por un programa C#, o en cualquier otra situación en la que se requiera que **struct** sin control contenga un arreglo. Los buffer de tamaño fijo sólo pueden utilizarse en un contexto inseguro.

Para crear un buffer de tamaño fijo, utiliza este formato de **fixed**:

```
fixed tipo nombre-buf[tamaño];
```

Aquí, *tipo* es el tipo de dato del arreglo, *nombre-buf* es el nombre del buffer de tamaño fijo y *tamaño* es la cantidad de elementos en el buffer. Los buffer de tamaño fijo pueden ser especificados sólo dentro de un **struct**.

Como mencionamos al principio de esta sección, la creación y el uso de código inseguro es un tema avanzado, y existen muchas más cuestiones relacionadas con su creación que las que se plantean aquí. Si escribir código inseguro va a formar parte de tu futuro en la programación, necesitarás estudiarlo más a fondo.

## Atributos

C# te permite añadir información declarativa a un programa en forma de un *atributo*. Un atributo define información adicional que está asociada con una clase, estructura, método y demás. Por ejemplo, puedes definir un atributo que determine el tipo de botón que una clase va a mostrar.

Los atributos se especifican entre corchetes, antecediendo al elemento sobre el que se aplicarán. Puedes definir tus propios atributos o utilizar los definidos por C#. Aunque crear tus propios atributos rebasa los alcances de este libro, es muy fácil utilizar dos de los atributos integrados en C#: **Condicional** y **Obsolete**. Ambos son examinados en las siguientes secciones.

## El atributo Condicional

El atributo **Condicional** es quizá el más interesante de C#. Te permite crear *métodos condicionales*. Un método condicional se invoca sólo cuando un símbolo específico ha sido definido a través de **#define**. De lo contrario, el método se ignora. De esta manera, un método condicional ofrece una alternativa a la compilación condicional que utiliza **#if**. Para utilizar el atributo **Condicional**, debes incluir la nomenclatura **System.Diagnostics**.

Comencemos con un ejemplo:

```
// Muestra el funcionamiento del atributo Condicional.  
#define ENSAYO  
  
using System;  
using System.Diagnostics;  
  
class Test {  
  
    [Conditional("ENSAYO")]  
    void Ensayo() { ← Ensayo() se ejecuta sólo si se define ENSAYO.  
        Console.WriteLine("Versión de ensayo, no está permitida su  
        distribución.");  
    }  
  
    [Conditional("LIBERA")]  
    void Libera() { ← Libera() se ejecuta sólo si se define LIBERA.  
        Console.WriteLine("Liberación de versión final.");  
    }  
}
```

```

static void Main() {
    Test t = new Test();

    t.Ensayo(); // se invoca sólo si ENSAYO es definido
    t.Libera(); // se invoca sólo si LIBERA es definido
}
}

```

Los datos generados por el programa son:

Versión de ensayo, no está permitida su distribución.

Veamos de cerca este programa para entender por qué se producen estos datos de salida. Primero observa que el programa define el símbolo **ENSAYO**. Después, observa cómo están codificados los métodos **Ensayo()** y **Libera()**. Ambos están antecedidos por el atributo **Condicional**, que tiene el siguiente formato general:

[Condicional símbolo]

donde *símbolo* es el símbolo que determina si el método será ejecutado. Este atributo puede ser utilizado sólo sobre métodos. Si el símbolo está definido, entonces cuando el método es invocado, éste se ejecuta. Si el símbolo no está definido, el método no será ejecutado.

Dentro de **Main()**, tanto **Ensayo()** como **Libera()** son invocados. Sin embargo, sólo se define **ENSAYO**. Por ello, **Ensayo()** se ejecuta. La invocación a **Libera()** se pasa por alto. Si defines **LIBERA**, entonces **Libera()** también será invocado. Si eliminas la definición de **ENSAYO**, entonces **Ensayo()** no será invocado.

Los métodos condicionales tienen unas cuantas restricciones: deben regresar **void**; deben ser miembros de una clase o estructura, no una interfaz; y no deben estar antecedidos por la palabra clave **override**.

## El atributo obsolete

El atributo **System.Obsolete** te permite marcar el elemento de un programa como obsoleto. Tiene dos formatos básicos. El primero es:

[Obsolete “mensaje”]

Aquí, *mensaje* se muestra cuando el elemento del programa se compila. He aquí un ejemplo breve:

```
// Muestra el atributo Obsolete.
```

```
using System;
```

```
public class Test {
```

```
[Obsolete("Usa MiMet2, en lugar de éste.")] ← Muestra una advertencia
public static int MiMet(int a, int b) {           si se usa MiMet().
    return a / b;
}
```

```
// Versión mejorada de MiMet.
public static int MiMet2(int a, int b) {
    return b == 0 ? 0 : a /b;
}

static void Main() {
    // Se muestra una advertencia.
    Console.WriteLine("4 / 3 es " + Test.MiMet(4, 3));

    // Aquí no hay advertencia.
    Console.WriteLine("4 / 3 es " + Test.MiMet2(4, 3));
}
```

Cuando se encuentra la invocación a **MiMet()** dentro de **Main()** durante la compilación del programa, se generará una advertencia que le indica al usuario que utilice **MiMet2()** en lugar del primero.

Un segundo formato de **Obsolete** se muestra a continuación:

[Obsolete (“mensaje”, error)]

Aquí, *error* es un valor booleano. Si es verdadero, entonces el uso del elemento obsoleto genera un error de compilación en lugar de una advertencia. Esta diferencia es, por supuesto, que un programa que contiene un error no puede ser compilado en un programa ejecutable.

## Operadores de conversión

En algunas situaciones, querrás utilizar el objeto de una clase en una expresión que involucra otros tipos de datos. En ocasiones, sobrecargar uno o más operadores puede proporcionar los medios para hacerlo. Sin embargo, en otros casos, bastará con una simple conversión de tipo: del tipo de la clase al tipo del objetivo. Para manejar estos casos, C# te permite crear un *operador de conversión*. Un operador de conversión convierte un objeto de tu clase a otro tipo.

Hay dos variedades de operadores de conversión: implícitos y explícitos. El formato general de cada uno se muestra a continuación:

```
public static operator implicit tipo-objetivo (tipo-fuente v) { return valor; }

public static operator explicit tipo-objetivo (tipo-fuente v) { return valor; }
```

Aquí, *tipo-objetivo* es el tipo al que se va a transformar el valor, *tipo-fuente* es el tipo a partir del cual se está realizando la transformación, y *valor* es el valor de la clase después de la conversión. Los operadores de conversión regresan datos del *tipo-objetivo*, y no se permite ningún otro especificador de regreso de tipo.

Si el operador de conversión se especifica como **implicit**, entonces la conversión se invoca automáticamente, como cuando un objeto se utiliza en una expresión con el tipo objetivo. Cuando el operador de conversión se especifica como **explicit**, la conversión se invoca cuando se utiliza una transformación. No puedes definir ambos tipos de conversiones, implícita y explícita, para los mismos tipos fuente y objetivo.

Para ilustrar el funcionamiento de los operadores de conversión, utilizaremos la clase **TresD** que fue creada en el capítulo 7. Recuerda que **TresD** almacena coordenadas tridimensionales. Su-

pongamos que quieras convertir un objeto de tipo **TresD** en un valor numérico con el fin de poder utilizarlo en una expresión numérica. Más aún, la conversión tendrá lugar al calcular la distancia del punto de origen, que será representado como un **double**. Para realizar la tarea, puedes utilizar un operador de conversión implícito como el siguiente:

```
public static implicit operator double(TresD op1)
{
    return Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z);
}
```

Toma un objeto **TresD** y regresa su distancia al origen como un valor **double**.

He aquí un programa que ilustra este operador de conversión:

```
// Un ejemplo que utiliza un operador de conversión implícita.
using System;

// Una clase de coordenadas tridimensionales.
class TresD {
    int x, y, z; // coordenadas 3-D

    public TresD() { x = y = z = 0; }
    public TresD(int i, int j, int k) { x = i; y = j; z = k; }

    // Sobrecarga binaria +.
    public static TresD operator +(TresD op1, TresD op2)
    {
        TresD result = new TresD();

        result.x = op1.x + op2.x;
        result.y = op1.y + op2.y;
        result.z = op1.z + op2.z;

        return result;
    }

    // Una conversión implícita de TresD a double.
    // Regresa la distancia del origen
    // al punto especificado.
    public static implicit operator double(TresD op1) ←
    {
        return Math.Sqrt(op1.x * op1.x + op1.y * op1.y + op1.z * op1.z);
    }

    // Muestra las coordenadas X, Y, Z.
    public void Show()
    {
        Console.WriteLine(x + ", " + y + ", " + z);
    }
}
```

Un operador de conversión de **TresD** a **double**.

```
class ConversionOpDemo {
    static void Main() {
        TresD alpha = new TresD(12, 3, 4);
        TresD beta = new TresD(10, 10, 10);
        TresD gamma = new TresD();
        double dist;

        Console.WriteLine("Aquí está alfa: ");
        alpha.Show();
        Console.WriteLine();
        Console.WriteLine("Aquí está beta: ");
        beta.Show();
        Console.WriteLine();

        // Suma alfa y beta. Esto NO invoca
        // el operador de conversión porque no es necesaria
        // una conversión a double.
        gamma = alfa + beta;
        Console.WriteLine("Resultado de alfa + beta: ");
        gamma.Show();
        Console.WriteLine();

        // La siguiente declaración invoca el operador
        // de conversión porque el valor de alfa es asignado
        // a dist, que es un double.
        dist = alfa; // convierte a double ←
        Console.WriteLine("Resultado de dist = alfa: " + dist);
        Console.WriteLine();

        // Esto también invoca al operador de conversión porque
        // la expresión requiere un valor double.
        if (beta > dist) ←
            Console.WriteLine("beta está más lejos del origen en comparación
                con alfa.");
    }
}
```

Operador de  
conversión  
invocado.

Los datos generados por el programa son:

Aquí está alfa: 12, 3, 4

Aquí está beta: 10, 10, 10

Resultado de alfa + beta: 22, 13, 14

Resultado de dist = alfa: 13

beta está más lejos del origen en comparación con alfa.

Como lo muestra el programa, cuando un objeto **TresD** es utilizado en una expresión **double**, como **dist = alfa**, la conversión se aplica al objeto. En este caso específico, la conversión regresa el valor 13, que es la distancia de **alfa** desde el origen. Sin embargo, cuando una expresión no requiere una conversión a **double**, el operador de conversión no es invocado. Por ello **gamma = alfa + beta** no invoca **operator double()**.

Recuerda que puedes crear diferentes operadores de conversión para satisfacer diferentes necesidades. Podrías definir uno que convierta a **long**, por ejemplo. Cada conversión se aplica automáticamente y de manera independiente.

Un operador de conversión implícita se usa automáticamente cuando se requiere la conversión en una expresión, cuando se transmite un objeto a un método, en una asignación y también cuando se ocupa una transformación explícita al tipo objetivo. De manera alternativa, puedes crear un operador de conversión explícito que es invocado sólo cuando se utiliza una transformación explícita. Un operador de conversión explícito no se invoca de manera automática. Por ejemplo, he aquí el operador de conversión del programa previo en una versión que utiliza la conversión explícita:

```
// Ahora es explícito.
public static explicit operator double(TresD opl)
{
    return Math.Sqrt(opl.x * opl.x + opl.y * opl.y + opl.z * opl.z);
}
```

Ahora, la siguiente declaración del ejemplo anterior

```
dist = alfa;
```

debe ser registrada para utilizar una transformación explícita, como se muestra aquí:

```
dist = (double) alfa;
```

Más aún, la siguiente declaración:

```
if(beta > dist)
    Console.WriteLine("beta está más lejos del origen en comparación con
                      alfa.");
```

debe ser actualizada así:

```
if((double) beta > dist)
    Console.WriteLine("beta está más lejos del origen en comparación con
                      alfa.");
```

Como el operador de conversión no está marcado como explícito, la conversión a **double** debe ser una transformación explícita en todos los casos.

Existen unas cuantas restricciones para los operadores de conversión:

- No puedes crear una conversión a partir de un tipo integrado a otro tipo integrado. Por ejemplo, no puedes redefinir la conversión de **double** a **int**.
- No puedes realizar una conversión desde ni hacia **object**.

- No puedes definir ambas conversiones, implícita y explícita, para los mismos tipos de fuente y objetivo.
- No puedes definir una conversión desde una clase base hacia las clases derivadas.
- No puedes definir una conversión desde ni hacia una interfaz.

## Pregunta al experto

**P:** **Como las conversiones implícitas se invocan automáticamente, sin la necesidad de transformación, ¿por qué querría crear una conversión explícita?**

**R:** Aunque muy convenientes, las conversiones implícitas deben ser utilizadas sólo en situaciones en las cuales la conversión está libre de errores de manera inherente. Para asegurarse de ello, las conversiones implícitas deben ser creadas sólo cuando se cumplan las siguientes dos condiciones. La primera es que no ocurra pérdida de información, como truncamiento, sobrecarga o pérdida de signo. La segunda es que la conversión no lance una excepción. Si la conversión no puede cumplir con estos dos requerimientos, entonces debes utilizar una conversión explícita.

## Una breve introducción a las colecciones

Una de las partes más importantes de .NET Framework son las colecciones. En lo que respecta a C#, una *colección* es un grupo de objetos. .NET Framework contiene una gran cantidad de interfaces y clases que definen e implementan varios tipos de colecciones. Las colecciones simplifican muchas tareas de programación porque aportan soluciones prefabricadas para varias estructuras de datos comunes, pero en ocasiones tediosas para desarrollar. Por ejemplo, existen colecciones integradas que dan soporte a arreglos dinámicos, listas vinculadas, pilas, colas y tablas hash.

Las colecciones API son muy grandes y contienen colecciones tanto genéricas como no genéricas. No es posible describir por completo su contenido o ilustrar su uso aquí. Sin embargo, como las colecciones se están convirtiendo en una parte fundamental para la programación con C#, son una característica de la que debes tener conocimiento. Ya cerca del final de este libro, esta sección proporciona una breve introducción a este importante subsistema. Conforme avances en tu estudio sobre C#, las colecciones son definitivamente una característica que querrás estudiar a fondo.

## Bases de las colecciones

El principal beneficio de las colecciones es que estandarizan la manera en que el programa maneja grupos de objetos. Todas las colecciones están diseñadas alrededor de un conjunto de interfaces claramente definido. Muchas implementaciones integrales de estas interfaces están prefabricadas, para que las utilices tal y como se presentan. También puedes implementar tus propias colecciones, pero rara vez necesitarás hacerlo.

Como se mencionó, .NET Framework define colecciones tanto genéricas como no genéricas. La versión 1.0 original contiene sólo versiones no genéricas, pero las colecciones genéricas fueron añadidas desde la versión 2.0. Aunque ambas versiones aún se utilizan, el nuevo código debe

enfocarse hacia las colecciones genéricas porque son de tipo seguro. (Las colecciones originales, no genéricas, almacenan referencias de objetos, lo que las hace vulnerables a los errores de inconsistencia de tipo.) Las colecciones de clases e interfaces no genéricas están declaradas en **System.Collections**. Las colecciones genéricas están declaradas en **System.Collections.Generic**. Como las colecciones genéricas han desplazado desde hace mucho a las no genéricas, son el único tipo que describimos aquí.

La funcionalidad básica de las colecciones se define por las interfaces que implementan. Para las colecciones genéricas, el fundamento es la interfaz **ICollection<T>**, que es implementada por todas las colecciones genéricas. Hereda **IEnumerable<T>** (que extiende **IEnumerable**) y define los métodos que se muestran aquí:

| Método                                                    | Descripción                                                                                                                                                                         |
|-----------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| void Add(T <i>obj</i> )                                   | Añade <i>obj</i> a la colección invocante.                                                                                                                                          |
| void Clear( )                                             | Elimina todos los elementos de la colección invocante.                                                                                                                              |
| bool Contains(T <i>obj</i> )                              | Regresa <b>true</b> si la colección invocante contiene el objeto transmitido en <i>obj</i> y <b>false</b> en caso contrario.                                                        |
| void CopyTo(T[ ] <i>objetivo</i> , int <i>inicioidx</i> ) | Copia el contenido de la colección invocante y el arreglo especificado en <i>objetivo</i> , comenzando con el índice especificado en <i>inicioidx</i> .                             |
| IEnumerator<T> GetEnumerator( )                           | Regresa el enumerador para la colección. (Especificado por <b>IEnumerable&lt;T&gt;</b> .)                                                                                           |
| IEnumerator GetEnumerator( )                              | Regresa el enumerador no genérico para la colección. (Especificado por <b>IEnumerable</b> .)                                                                                        |
| bool Remove(T <i>obj</i> )                                | Elimina la primera aparición de <i>obj</i> de la colección invocante. Regresa <b>true</b> si <i>obj</i> fue eliminado y <b>false</b> si no fue encontrado en la colección invocada. |

Los métodos que modifican una colección lanzarán una excepción **NotSupportedException** si la colección es sólo-lectura.

**ICollection<T>** también define las siguientes propiedades:

```
int Count { get; }

bool IsReadOnly { get; }
```

**Count** contiene la cantidad de elementos que actualmente se encuentran en la colección. **IsRead-Only** es verdadero si la colección es sólo-lectura. Es falso si la colección es lectura/escritura.

Dado que **ICollection<T>** hereda las interfaces **IEnumerable<T>**, asegura que todas las colecciones de clases puedan ser enumeradas (recorridas un elemento a la vez). Más aún, la herencia de **IEnumerable<T>** permite que las colecciones puedan ser utilizadas como fuentes de datos para queries o para reiteraciones del loop **foreach**. (Recuerda que sólo instancias de objetos que implementan **IEnumerable** o **IEnumerable<T>** pueden utilizarse como fuentes de datos para un query.) Dado que las colecciones implementan **IEnumerable<T>**, también soportan los métodos de extensión definidos por **IEnumerable<T>** (ver capítulo 14).

Las colecciones API definen muchas otras interfaces para añadir funcionalidad. Por ejemplo, **IList<T>** extiende **ICollection<T>**, añadiendo soporte a colecciones cuyos elementos puedan ser accedidos a través de un índice. **IDictionary<TK, TV>** extiende **ICollection<T>** para dar soporte al almacenamiento de pares clave/valor.

Las colecciones API proporcionan muchas implementaciones de las colecciones de interfaces. Por ejemplo, la colección genérica **List<T>** implementa un arreglo dinámico de tipo seguro, que es un arreglo que crece conforme sea necesario. Hay clases que implementan pilas y colas, como **Stack<T>** y **Queue<T>**, respectivamente. Otras clases, como **Dictionary<TK, TV>**, almacenan pares de clave/valor.

Aunque no es posible hacer un análisis profundo de cada interfaz de colección, a continuación presentamos un caso de estudio que te dará una idea de su poder e ilustra la manera general de su uso. El ejemplo utiliza la colección **List<T>**.

## Un caso de estudio de colecciones: crear un arreglo dinámico

Tal vez la colección más utilizada es **List<T>**, que implementa un arreglo dinámico genérico. Contiene los constructores que se presentan a continuación:

```
public List()
public List(IEnumerable<T>c)
public List(int capacidad)
```

El primer constructor compone una lista vacía con la capacidad inicial por defecto. El segundo construye una lista que es inicializada con elementos de la colección especificada por *c* y con una capacidad inicial igual a la cantidad de elementos del arreglo. El tercero construye una lista que tiene la *capacidad* inicial especificada. La capacidad crece automáticamente conforme se añaden los elementos a **List<T>**. Cada vez que la lista debe ser incrementada, su capacidad crece.

**List<T>** implementa varias interfaces, incluyendo **IList<T>**. La interfaz **IList<T>** extiende **ICollection<T>**, **IEnumerable<T>** e **IEnumerable**. La interfaz **IList<T>** define el comportamiento de una colección genérica, la cual permite que los elementos sean accesados a través de un índice de base cero. Además de los métodos especificados por las interfaces que extiende, **IList<T>** añade los métodos que se muestran a continuación. Si la colección es sólo-lectura, entonces los métodos **Insert()** y **RemoveAt()** lanzarán una excepción **NotSupportedException**.

| Método                             | Descripción                                                                                                                               |
|------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| int <b>IndexOf(T obj)</b>          | Regresa el índice de <i>obj</i> si <i>obj</i> está contenido dentro de la colección invocante. Si <i>obj</i> no se encuentra, regresa -1. |
| void <b>Insert(int idx, T obj)</b> | Inserta <i>obj</i> en el índice especificado por <i>idx</i> .                                                                             |
| void <b>RemoveAt(int idx)</b>      | Elimina el objeto en el índice especificado por <i>idx</i> de la colección invocadora.                                                    |

**IList<T>** define el siguiente indexador:

```
T this[int idx] {get; set;}
```

Este indexador establece u obtiene el valor del elemento que se encuentra en el índice especificado por *idx*.

Además de la funcionalidad definida por las interfaces que implementa, **List<T>** proporciona mucho de sí misma. Por ejemplo, aporta métodos que ordenan una lista, realizan búsquedas binarias y convierten una lista en un arreglo. **List<T>** también define una propiedad llamada **Capacity** que establece u obtiene la capacidad de la lista invocante. La capacidad es la cantidad de elementos que pueden almacenar antes que la lista deba ser alargada. (No es la cantidad de elementos que actualmente contiene la lista.) Como la lista crece de manera automática, no es necesario establecer la capacidad manualmente. Sin embargo, por razones de eficiencia, tal vez quieras establecer la capacidad si sabes por adelantado cuántos elementos contendrá la lista. Esto previene la saturación asociada con la reserva de más memoria.

He aquí el programa que muestra el funcionamiento básico de **List<T>**. Crea un arreglo dinámico de tipo **int**. Observa que la lista se expande y se contrae automáticamente, con base en la cantidad de elementos que contiene.

```
// Muestra el funcionamiento de List<T>.

using System;
using System.Collections.Generic;

class ListDemo {
    static void Main() {
        // Crea una lista de enteros.
        List<int> lst = new List<int>(); ← Crea una instancia de List para int.

        Console.WriteLine("Cantidad inicial de elementos: " + lst.Count);

        Console.WriteLine();

        Console.WriteLine("Añade 5 elementos");
        // Añade elementos a la lista de arreglos.
        lst.Add(1);
        lst.Add(-2);
        lst.Add(14); ← Añade elementos a la lista.
        lst.Add(9);
        lst.Add(88);

        Console.WriteLine("Cantidad de elementos: " + lst.Count);

        // Muestra la lista del arreglo utilizando la indexación del mismo.
        Console.Write("Contiene: ");
        for(int i=0; i < lst.Count; i++)
            Console.Write(lst[i] + " "); ← Indexa la lista.
        Console.WriteLine("\n");

        Console.WriteLine("Elimina 2 elementos");
        // Elimina elementos de la lista del arreglo.
        lst.Remove(-2); ← Elimina elementos de la lista.
        lst.Remove(88);

        Console.WriteLine("Cantidad de elementos: " + lst.Count);
```

## 582 Fundamentos de C# 3.0

---

```
// Usa el loop foreach para mostrar la lista.  
Console.WriteLine("Contiene: ");  
foreach(int i in lst) ←———— Hace un ciclo por la lista con el loop foreach.  
    Console.Write(i + " ");  
    Console.WriteLine("\n");  
  
Console.WriteLine("Añade 5 elementos");  
// Añade suficientes elementos para hacer que la lista crezca.  
for(int i=0; i < 5; i++) ←———— Expande la lista.  
    lst.Add(i);  
  
Console.WriteLine("Cantidad de elementos después de añadir 5: " +  
                  lst.Count);  
Console.WriteLine("Contiene: ");  
foreach(int i in lst)  
    Console.Write(i + " ");  
    Console.WriteLine("\n");  
  
// Cambia el contenido utilizando la indexación del arreglo.  
Console.WriteLine("Cambia los primeros tres elementos");  
lst[0] = -10;  
lst[1] = -lst[1]; ←———— Cambia la lista.  
lst[2] = 99;  
  
Console.WriteLine("Contiene: ");  
foreach(int i in lst)  
    Console.Write(i + " ");  
    Console.WriteLine();  
}  
}
```

Los datos generados por el programa son:

```
Cantidad inicial de elementos: 0  
  
Añade 5 elementos  
Cantidad de elementos: 5  
Contiene: 1 -2 14 9 88  
  
Elimina 2 elementos  
Cantidad de elementos: 3  
Contiene: 1 14 9  
  
Añade 10 elementos  
Cantidad de elementos después de añadir 10: 13  
Contiene: 1 14 9 0 1 2 3 4 5 6 7 8 9  
  
Cambia los primeros tres elementos  
Contiene: -10 -14 99 0 1 2 3 4 5 6 7 8 9
```

El programa principia creando una instancia de **List<int>** llamada **lst**. Al principio esta colección está vacía. Observa cómo crece su tamaño conforme se añaden elementos. Como ya se explicó, **List<T>** crea un arreglo dinámico, el cual crece conforme es necesario para acomodar la cantidad de elementos que debe contener. También observa cómo **lst** puede ser indexada utilizando la misma sintaxis que se ocupa para indexar un arreglo.

Como la colección **List<T>** crea un arreglo dinámico e indexable, suele utilizarse en lugar de un arreglo. La principal ventaja es que no necesitas saber cuántos elementos serán almacenados en la lista en el momento de la compilación. Por supuesto, los arreglos ofrecen mayor eficiencia relativamente, por lo que **List<T>** cambia velocidad por conveniencia.

## Prueba esto

## Utilizar la colección **Queue<T>**

En los capítulos anteriores, varios ejemplos *Prueba esto* han desarrollado y evolucionado una clase de orden en cola como medio para ilustrar diferentes conceptos fundamentales de la programación en C#, como encapsulación, propiedades, excepciones y demás. Aunque crear tus propias estructuras de datos, como el orden en cola, es un buen método para aprender C#, no es algo que normalmente necesites hacer. En lugar de ello, por lo regular utilizarás una de las colecciones estándar. En el caso del orden en cola, se trata de **Queue<T>**, que proporciona una implementación de alto rendimiento que está totalmente integrada en el marco de las colecciones generales. En esta sección, el ejercicio final de *Prueba esto*, aprenderás a poner **Queue<T>** en acción. Crea un pequeño programa que simula utilizar una cola para garantizar el acceso a los usuarios de una red.

**Queue<T>** es una colección dinámica que crece conforme es necesario para acomodar los elementos que debe almacenar. **Queue<T>** define los siguientes constructores:

```
public Queue()
public Queue(int capacidad)
public Queue(IEnumerable<T>c)
```

El primero crea una cola vacía con la capacidad inicial por defecto. El segundo crea una cola vacía con capacidad inicial especificada por *capacidad*. El tercero crea una cola que contiene los elementos de la colección especificada por *c*.

Además de la funcionalidad definida por la colección de interfaces que implementa, **Queue<T>** define los métodos que se muestran a continuación. Para colocar un objeto en la cola, invoca **Enqueue()**. Para eliminar y regresar el objeto al inicio de la cola, invoca **Dequeue()**. Se lanzará una excepción **InvalidOperationException** si invocas **Dequeue()** cuando la cola invocante está vacía. Puedes utilizar **Peek()** para regresar, sin eliminar, el siguiente objeto.

(continúa)

| Método                    | Descripción                                                                                                     |
|---------------------------|-----------------------------------------------------------------------------------------------------------------|
| public T Dequeue( )       | Regresa el objeto al inicio de la cola invocante. El objeto es eliminado en el proceso.                         |
| public void Enqueue(T v)  | Añade v al final de la cola.                                                                                    |
| public T Peek( )          | Regresa el objeto al inicio de la cola invocante, pero no lo elimina.                                           |
| public T[ ] ToArray( )    | Regresa un arreglo que contiene copias de los elementos de la cola invocante.                                   |
| public void TrimExcess( ) | Elimina el exceso de capacidad de la cola invocante si el tamaño es menor que el 90 por ciento de su capacidad. |

Este ejemplo utiliza una cola para simular el registro de acceso a una red por parte de una colección de usuarios. En realidad no hace ningún registro. En vez de ello, simplemente rellena una cola con nombres de usuario y después concede el acceso de los usuarios con base en el orden en el cual entraron a la cola. Por supuesto, como se trata de una simulación, el programa simplemente muestra el nombre del usuario cuando se le concede el acceso.<sup>1</sup>

## Paso a paso

1. Comienza creando la simulación como se muestra aquí:

```
// Utiliza la clase Queue<T> para simular el registro de acceso de
usuarios a una red.

using System;
using System.Collections.Generic;

class QueueDemo {
    static void Main() {
        Queue<string> userQ = new Queue<string>();
```

Observa que se crea una **Queue** que invoca **userQ** para que contenga referencias a objetos de tipo **string**.

2. Añade el siguiente código que coloca los nombres de usuario en la cola:

```
Console.WriteLine("Añade usuarios a la cola de ingreso a la red.\n");
userQ.Enqueue("Eric");
userQ.Enqueue("Tom");
userQ.Enqueue("Ralph");
userQ.Enqueue("Ken");
```

3. Añade el código que elimina un nombre a la vez, el cual simula el acceso garantizado a la red.

```
Console.WriteLine("Asegura el acceso a la red en el orden de la cola.
\n");
```

---

<sup>1</sup> Si estás utilizando la versión 3.5 o superior de Microsoft Visual C# 2008 Express, es **indispensable** que agregues la referencia **System** a C# para que funcione el siguiente ejemplo. Haz clic derecho sobre el nombre del programa en la ventana “Explorador de soluciones”. Selecciona “Agregar referencia”. Aparecerá una ventana emergente. En la pestaña “.NET” haz clic sobre la referencia **System**. Finalmente haz clic en “Aceptar”.

```
while(userQ.Count > 0) {
    Console.WriteLine("Acceso a la red garantizado para : " + userQ.
        Dequeue());
}
```

**4.** Finaliza el programa.

```
    Console.WriteLine("\nLa cola de usuarios terminó.");
}
}
```

**5.** A continuación presentamos el programa completo que utiliza **Queue<T>** para simular el registro de usuarios que accesan a una red:

```
// Utiliza la clase Queue<T> para simular el registro de acceso de
usuarios a una red.

using System;
using System.Collections.Generic;

class QueueDemo {
    static void Main() {
        Queue<string> userQ = new Queue<string>();

        Console.WriteLine("Añade usuarios a la cola de ingreso a la red.
\n");

        userQ.Enqueue("Eric");
        userQ.Enqueue("Tom");
        userQ.Enqueue("Ralph");
        userQ.Enqueue("Ken");

        Console.WriteLine("Asegura el acceso a la red en el orden de la
cola.\n");

        while(userQ.Count > 0) {
            Console.WriteLine("Acceso a la red garantizado para : " + userQ.
                Dequeue());
        }

        Console.WriteLine("\nLa cola de usuarios terminó.");
    }
}
```

Los datos que genera el programa son:

Añade usuarios a la cola de ingreso a la red.

Asegura el acceso a la red en el orden de la cola.

Acceso a la red garantizado para: Eric

Acceso a la red garantizado para: Tom

Acceso a la red garantizado para: Ralph

Acceso a la red garantizado para: Ken

La cola de usuarios terminó.

(continúa)

Un punto clave de este ejemplo es cómo se necesita poco código para implementar la simulación. Si tuvieras que desarrollar este programa por ti mismo (como sucedió en las anteriores secciones Prueba esto), el código hubiera sido mucho más. Más aún, la clase estándar **Queue<T>** ofrece una solución que todos los programadores de C# reconocerán de manera instantánea, lo que hace que tu código sea más fácil de mantener.

## Otras palabras clave

Para concluir este libro, explicamos brevemente las restantes palabras clave definidas por C#, que no han sido mencionadas en los anteriores capítulos.

### El modificador de acceso interno

Además de los modificadores de acceso **public**, **private** y **protected**, que hemos utilizado a lo largo de este libro, C# también define **internal**. Mencionado brevemente en el capítulo 6, **internal** declara que un miembro es conocido a través de todos los archivos en un ensamblado, pero desconocido fuera del mismo. Un *ensamblado* es un archivo (o archivos) que contiene toda la información de desarrollo y versiones de un programa. Así, en términos sencillos, un miembro marcado como **internal** es conocido por todo el programa, pero en ningún otro lugar.

### sizeof

En ocasiones, es posible que encuentres de utilidad saber el tamaño, en bytes, de un tipo valor de C#. Para obtener esta información utiliza el operador **sizeof**. Tiene el siguiente formato general:

`sizeof (tipo)`

Aquí, *tipo* es el tipo cuyo tamaño va a ser obtenido. De esta manera, su intención principal es para situaciones de caso especial, en particular cuando se trabaja con una mezcla de código controlado y sin control.

### lock

La palabra clave **lock** se utiliza cuando se trabaja con *hilos múltiples*. En C#, un programa puede contener dos o más *hilos de ejecución*. Cuando esto sucede, piezas del programa pasan por un proceso de multitarea. Así, piezas del programa se ejecutan de manera independiente y simultánea. Esto hace que surja la posibilidad de un problema: ¿qué sucedería si dos hilos intentaran utilizar un recurso que puede ser utilizado sólo por un hilo a la vez? Para resolver este problema, puedes crear una *sección de código crítico* que será ejecutado por uno y sólo un hilo a la vez. Esta tarea se realiza con **lock**. Su formato general se muestra aquí:

```
lock(obj) {  
    // sección crítica  
}
```

Aquí, *obj* es el objeto sobre el cual se sincroniza el candado (lock). Si un hilo ya ha entrado en la sección crítica, entonces el segundo hilo tendrá que esperar hasta que el primero abandone esa sección.

Cuando el primer hilo abandona la sección crítica, se abre el candado y el segundo hilo tiene acceso garantizado; en ese punto el segundo hilo puede ejecutar la sección crítica.

## readonly

Puedes crear un campo sólo-lectura en una clase al declararlo como **readonly**. A un campo **readonly** se le puede dar un valor sólo a través del uso de un inicializador cuando es declarado, o bien asignándole un valor dentro de un constructor. Una vez que el valor ha sido establecido, no puede ser cambiado fuera del constructor. Así, los campos **readonly** son una buena manera para crear constantes, como las dimensiones de un arreglo, que se utilizarán a lo largo del programa. Están permitidos campos **readonly** tanto estáticos como no estáticos.

He aquí un ejemplo que crea y utiliza un campo **readonly**:

```
// Muestra el uso de readonly.
using System;

class MiClase {
    public static readonly int TAMAÑO = 10; ← En esencia, esto declara una constante.
}

class DemoReadOnly {
    static void Main() {
        int[] nums = new int[MiClase.TAMAÑO];

        for(int i=0; i < MiClase.TAMAÑO; i++)
            nums[i] = i;

        foreach(int i in nums)
            Console.Write(i + " ");

        // MiClase.TAMAÑO = 100; // !!!Error!!! no puede cambiar
    }
}
```

Aquí, **MiClase.TAMAÑO** se inicializa con 10. Después de eso puede ser utilizada, pero no cambiada. Para probarlo, intenta eliminar los símbolos de comentario al inicio de la última línea y luego recompila el programa. Verás que da como resultado un error.

## stackalloc

Puedes reservar memoria de la pila utilizando **stackalloc**. Puede ser utilizada sólo cuando se inicializan variables locales y tiene el siguiente formato general:

*tipo*\**p* = stackalloc *tipo*[*tamaño*]

Aquí, *p* es un puntero que recibe la dirección de la memoria que es lo suficientemente grande para contener la cantidad de objetos *tipo* especificada por *tamaño*. **stackalloc** debe utilizarse en un contexto inseguro.

Normalmente, la memoria necesaria para objetos es reservada del *cúmulo*, que es una región de memoria libre. Reservar memoria de la pila es la excepción. Las variables localizadas en la pila

no son eliminadas por el recolector de basura. En lugar de ello, existen sólo mientras se ejecuta el bloque de código en el cual están declaradas. La única ventaja de utilizar **stackalloc** es que no necesitas preocuparte de que esas variables sean movidas por el recolector de basura.

## La declaración **using**

Además de la *directiva using* explicada anteriormente, **using** tiene un segundo formato que recibe el nombre de *declaración using*. Tiene los siguientes formatos generales:

```
using (obj) {
    // utiliza obj
}

using (tipo obj = inicializador) {
    // utiliza obj
}
```

Aquí, *obj* es una expresión que debe ser evaluada contra un objeto que implemente la interfaz **System.IDisposable**. Especifica una variable que será utilizada dentro del bloque **using**. En el primer formato, el objeto es declarado fuera de la declaración **using**. En el segundo, el objeto es declarado dentro de la declaración **using**. Cuando el bloque concluye, el método **Dispose()** (definido por la interfaz **System.IDisposable**) hará una invocación sobre *obj*. Así, una declaración **using** proporciona un medio a través del cual los objetos serán eliminados automáticamente cuando ya no se necesiten. Recuerda, la declaración **using** aplica sólo a objetos que implementan la interfaz **System.IDisposable**.

A continuación presentamos un ejemplo de ambos formatos de la declaración **using**:

```
// Muestra el uso de la declaración using.

using System;
using System.IO;

class UsingDemo {
    static void Main() {
        StreamReader sr = new StreamReader("test.txt");

        // Utiliza el objeto dentro de la declaración using.
        using(sr) {
            // ...
        }

        // Crea StreamReader dentro de la declaración using.
        using(StreamReader sr2 = new StreamReader("test.txt")) {
            // ...
        }
    }
}
```

La clase **StreamReader** implementa la interfaz **IDisposable** (a través de su clase base **TextReader**). Por ello puede ser utilizada en una declaración **using**. Cuando la declaración **using** finaliza, **Dispose()** es invocada automáticamente sobre las variables de flujo, y en consecuencia el flujo es cerrado.

## const y volatile

El modificador **const** es utilizado para declarar campos o variables locales que no pueden alterarse. A estas variables debe dárseles un valor inicial cuando son declaradas. Así, una variable **const** es esencialmente una constante. Por ejemplo,

```
const int i = 10;
```

crea una variable **const** llamada **i** con un valor de 10.

El modificador **volatile** le indica al compilador que el valor de un campo puede ser alterado por dos o más hilos que se están ejecutando actualmente. En esta situación, es posible que un hilo no sepa cuándo ha sido modificado el campo por otro hilo. Esto es importante, porque el compilador de C# realizará en automático ciertas optimizaciones que funcionarán sólo cuando el campo sea accesado por un solo hilo en ejecución. Para prevenir que tales optimizaciones sean aplicadas a un campo compartido, se declara como **volatile**. Esto indica al compilador que debe obtener el valor de este campo cada vez que es accesado.

## El modificador partial

El modificador **partial** tiene dos usos. Primero, puede utilizarse para permitir que una definición de clase, estructura o interfaz sea dividida en dos o más piezas, cada una de las cuales puede residir en un archivo separado. Cuando tu programa se compila, las piezas de la clase, estructura o interfaz se unen, formando el tipo completo. Segundo, en una clase o estructura parcial, **partial** puede utilizarse para permitir que la declaración de un método sea separada de su implementación. Cada uno de estos usos se explica a continuación.

### Tipos parciales

Cuando se usa para crear un tipo parcial, el modificador **partial** tiene el siguiente formato general:

```
partial class nombretipo { // ... }
```

Aquí, *nombretipo* es el nombre de la clase, estructura o interfaz que va a ser dividida en piezas. Cada parte del tipo parcial debe ser modificada por **partial**.

He aquí un ejemplo que divide una sencilla clase de coordenadas XY en tres archivos separados. El primer archivo se muestra a continuación:

```
partial class XY {
    public XY(int a, int b) {
        X = a;
        Y = b;
    }
}
```

El segundo archivo es:

```
partial class XY {  
    public int X { get; set; }  
}
```

El tercer archivo es:

```
partial class XY {  
    public int Y { get; set; }  
}
```

El siguiente archivo muestra el uso de **XY**:

```
// Muestra las definiciones parciales de clase.  
using System;  
  
class Test {  
    static void Main() {  
        XY xy = new XY(1, 2);  
  
        Console.WriteLine(xy.X + ", " + xy.Y);  
    }  
}
```

Para utilizar **XY**, todos los archivos deben incluirse en el compilador. Por ejemplo, suponiendo que los archivos **XY** sean llamados **xy1.cs**, **xy2.cs**, **xy3.cs**, y que la clase **Test** esté contenida en un archivo llamado **test.cs**, entonces compila **Test** utilizando la siguiente línea de comando:

```
csc test.cs xy1.cs xy2.cs xy3.cs
```

Un último punto: es legal tener clases genéricas parciales. Sin embargo, los parámetros de tipo de cada declaración parcial deben coincidir con las otras partes.

## Métodos parciales

Dentro de un tipo parcial que es una clase o una estructura, puedes utilizar **partial** para crear un método parcial. Un método parcial tiene su declaración en una parte y su implementación en otra. Los métodos parciales fueron añadidos en la versión 3.0 de C#.

El aspecto clave de los métodos parciales es que ¡no se requiere implementación! Cuando un método parcial no es implementado por otra parte de la clase o estructura, entonces todas las invocaciones al método parcial son ignoradas silenciosamente. Esto hace posible que la clase especifique, pero no requiera, funcionalidad opcional. Si esa funcionalidad no se implementa, entonces simplemente se ignora.

A continuación presentamos una versión extendida del programa anterior que crea un método parcial llamado **Muestra()**. Es invocado por otro método llamado **MuestraXY()**.

```
// Muestra el funcionamiento de un método parcial.  
using System;
```

```

partial class XY {
    public XY(int a, int b) {
        X = a;
        Y = b;
    }

    // Declara un método parcial.
    partial void Muestra();
}

partial class XY {
    public int X { get; set; }

    // Implementa un método parcial.
    partial void Muestra() {
        Console.WriteLine("{0}, {1}", X, Y);
    }
}

partial class XY {
    public int Y { get; set; }

    // Invoca un método parcial.
    public void MuestraXY() {
        Muestra();
    }
}

class Test {
    static void Main() {
        XY xy = new XY(1, 2);

        xy.MuestraXY();
    }
}

```

Observa que **Muestra()** está declarada en una parte de **XY** y es implementada en otra parte. La implementación muestra los valores de **X** y **Y**. Esto significa que cuando **Muestra()** es invocada por **MuestraXY()**, la invocación tiene efecto y, de hecho, mostrará los valores de **X** y **Y**. Sin embargo, si comentas la implementación de **Muestra()**, entonces la invocación de **Muestra()** dentro de **MuestraXY()** no hará nada.

Los métodos parciales tienen varias restricciones, entre las que se incluyen las siguientes: deben ser **void**; no deben tener modificadores de acceso; no pueden ser virtuales; y no pueden utilizar parámetros **out**.

## yield

La palabra clave contextual **yield** se usa dentro de un reiterador, que es un método, operador o accesador que regresa los miembros de un conjunto de objetos, un elemento a la vez, en secuencia. Los reiteradores se utilizan generalmente con colecciones.

## extern

La palabra clave **extern** tiene dos usos. Primero, indica que un método es proveído por código externo, que por lo general es código sin control. Segundo, se usa para crear un alias para un ensamblado externo.

## ¿Qué sigue?

¡Felicitaciones! Si has leído y trabajado a lo largo de estos 15 capítulos, entonces ya puedes llamarlo un programador de C#. Por supuesto, todavía tienes muchas, muchas cosas por aprender sobre C#, sus bibliotecas y subsistemas, pero ahora ya tienes cimientos sólidos sobre los cuales puedes construir tu conocimiento y experiencia. A continuación presentamos algunos temas sobre los que querrás profundizar:

- Crear aplicaciones multilectura.
- Utilizar Windows Forms.
- Utilizar las colecciones de clases.
- Trabajo en red con C# y .NET.

Para continuar tu estudio sobre C#, te recomendamos el libro *C# 3.0: The Complete Reference* (McGraw-Hill).



## Autoexamen Capítulo 15

1. Menciona las directivas de procesador que se utilizan para la compilación condicional.
2. ¿Qué acciones realiza **#elif**?
3. ¿Cómo puedes obtener una instancia **System.Type** que represente el tipo del objeto en tiempo de ejecución?
4. ¿Qué es **is**?
5. ¿Qué beneficios proporciona **as**?
6. Muestra cómo declarar un **int** anulable llamado **cuenta** que se inicializa con **null**.
7. Un tipo anulable puede representar todos los valores de su tipo subyacente, además del valor **null**. ¿Cierto o falso?
8. ¿Qué invoca el operador **??** y qué acciones realiza?
9. ¿Qué es el código inseguro?
10. Muestra cómo declarar un puntero a **double**. Llama al puntero **ptr**.
11. Muestra la sintaxis de atributo.

- 12.** ¿Qué son los dos tipos de operadores de conversión? Muestra sus formatos generales.
- 13.** ¿Qué es una colección? ¿Qué nomenclaturas están en las colecciones genéricas?
- 14.** ¿Qué interfaz es implementada por todas las colecciones genéricas?
- 15.** ¿Qué palabra clave se utiliza para declarar una clase parcial? ¿Qué palabra clave se utiliza para declarar un método parcial?
- 16.** Por cuenta propia, continúa explorando y experimentando con C#.



# Apéndice A

Respuestas a los  
autoexámenes

## Capítulo 1: Fundamentos de C#

1. MSIL significa Microsoft Intermediate Language (Lenguaje Intermediario de Microsoft). Se trata de un conjunto optimizado y manejable de instrucciones de lenguaje ensamblador que es compilado en código ejecutable por un compilador JIT. MSIL ayuda a que C# sea un lenguaje adaptable, seguro y que tenga compatibilidad para la mezcla de lenguajes.
2. El Lenguaje Común de Tiempo de Ejecución (Common Language Runtime, CLR) es parte de .NET que controla la ejecución de C# y otros programas compatibles con .NET.
3. Encapsulación, polimorfismo y herencia.
4. Los programas C# comienzan a ejecutarse en Main().
5. Una variable es una locación de memoria con nombre. El contenido de la variable puede modificarse durante la ejecución del programa. Una nomenclatura es una región declarativa. Las nomenclaturas ayudan a mantener un conjunto de nombres separado de otros.
6. Las variables inválidas son B y D. Los nombres de variable no pueden comenzar con un \$ o un dígito.
7. Un comentario de una sola línea inicia con // y termina junto con la línea. Un comentario multilínea comienza con /\* y termina con \*/.
8. El formato general de if es:

if (*condición*) *declaración*;

El formato general de for es:

for(*inicialización*; *condición*; *reiteración*) *declaración*;

9. Un bloque de código inicia con { y termina con }.

10. No, using System no es necesario, pero dejarlo fuera significa que debes calificar completamente los miembros de la nomenclatura System colocando la palabra System antes del nombre.

11. // Calcula el peso en la Luna.

```
using System;

class Moon {
    static void Main() {
        double pesoterra; // peso en la Tierra
        double pesoluna; // peso en la Luna

        pesoterra = 75.0;
        pesoluna = pesoterra * 0.17;

        Console.WriteLine(pesoterra + " libras terrestres son equivalentes
                           a " + pesoluna + " libras en la Luna.");
    }
}
```

```
12. /*  
Este programa muestra una tabla de  
conversión de pulgadas a metros.  
*/  
  
using System;  
  
class TablaPulgadaAMetro {  
    static void Main() {  
        double pulgadas, metros;  
        int contador;  
  
        contador = 0;  
        for(pulgadas = 1.0; pulgadas <= 144.0; pulgadas++) {  
  
            // Convierte a metros.  
            metros = pulgadas / 39.37;  
  
            Console.WriteLine(pulgadas + " pulgadas son " + metros + "  
metros.");  
  
            contador++;  
  
            // Cada 12a línea, coloca una línea en blanco.  
            if(contador == 12) {  
                Console.WriteLine();  
                contador = 0; // reajusta el contador de líneas  
            }  
        }  
    }  
}
```

## Capítulo 2: Introducción a los tipos de datos y operadores

- 1.** C# especifica estrictamente el rango y comportamiento de sus tipos simples para asegurar la portabilidad e interoperabilidad en el ambiente de lenguaje mezclado.
- 2.** El tipo de caracteres de C# es char. Los caracteres de C# son Unicode en lugar de ASCII, que es utilizado por muchos otros lenguajes de computación.
- 3.** Falso. Un valor bool debe ser true o false.
- 4.** Console.WriteLine ("Uno\nDos\nTres");
- 5.** Hay tres fallas fundamentales en el fragmento. Primero, sum se crea cada vez que el bloque creado por el loop for es ingresado y se destruye cuando termina. De esta manera, no mantendrá su valor entre reiteraciones. Intentar utilizar sum para conservar una suma constante de reiteraciones no tiene sentido. Segundo, sum no será conocida fuera del bloque donde es declarada. Así, la referencia que se hace a ella en la declaración WriteLine( ) es inválida. Tercero, no se ha dado a sum un valor inicial.

**6.** Cuando el operador de incremento antecede a su operando, C# incrementará el operando antes de obtener su valor. Si el operador se localiza después del operando, entonces C# primero obtendrá el valor del operando, luego su valor será incrementado.

**7.** if((b != 0) && (val / b)) ...

**8.** En una expresión, byte y short son promovidos a int.

**9.** A.

**10.** Se necesita una transformación cuando se hace una conversión entre tipos incompatibles o cuando ocurre una conversión de estrechamiento.

**11.** He aquí una manera de encontrar los primos entre 2 y 100. Por supuesto, existen otras soluciones.

```
// Encuentra los números primos entre 2 y 100.

using System;

class Primos {
    static void Main() {
        int i, j;
        bool esprimo;

        for(i=2; i < 100; i++) {
            esprimo = true;

            // Prueba si un número es uniformemente divisible.
            for(j=2; j <= i/j; j++)
                // Si i es uniformemente divisible, entonces no es primo.
                if((i%j) == 0) esprimo = false;

            if(esprimo)
                Console.WriteLine(i + " es primo.");
        }
    }
}
```

## Capítulo 3: Declaraciones para el control del programa

**1.** // Cuenta espacios.

```
using System;

class Espacios {
    static void Main() {
        char ch;
        int espacios = 0;

        Console.WriteLine("Escribe un punto para detener.");
    }
}
```

```
do {
    ch = (char) Console.Read();
    if(ch == ' ') espacios++;
} while(ch != '.');

Console.WriteLine("Espacios: " + espacios);
}
```

**2.** No. La regla de “no intromisión” declara que la secuencia de código de una etiqueta case no debe continuar a la siguiente. Sin embargo, las etiquetas pueden “apilarse”.

**3.** if(*condición*)

*declaración*;

else if (*condición*)

*declaración*;

else if (*condición*)

*declaración*;

.

.

.

else

*declaración*;

**4.** El último else asociado con el if externo, que es el if más cercano al mismo nivel del else.

**5.** for(int i = 1000; i >= 0; i -= 2) // ...

**6.** No, i no es conocida fuera del loop for en el cual se declara.

**7.** Un break provoca la finalización del loop o declaración switch más cercana que lo encierra.

**8.** Después de que se ejecuta break, se muestra “después del while”.

**9.** 0 1  
2 3  
4 5  
6 7  
8 9

**10.** /\* Usa un loop for para generar la progresión

```
1 2 4 8 16, ...
*/
```

```
using System;
```

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

```
        for(int i = 1; i < 100; i += i)
            Console.WriteLine(i + " ");
    }
}

11. // Cambia mayúsculas.

using System;

class CambiaMay {
    static void Main() {
        char ch;
        int cambios = 0;

        Console.WriteLine("Escribe un punto para detener.");

        do {
            ch = (char) Console.Read();
            if(ch >= 'a' && ch <= 'z') {
                ch -= (char) 32;
                cambios++;
                Console.WriteLine(ch);
            }
            else if(ch >= 'A' && ch <= 'Z') {
                ch += (char) 32;
                cambios++;
                Console.WriteLine(ch);
            }
        } while(ch != '.');
        Console.WriteLine("Cambia mayúsculas: " + cambios);
    }
}
```

## Capítulo 4: Introducción a las clases, objetos y métodos

- 1.** Una clase es una abstracción lógica que describe la forma y el comportamiento de un objeto. Un objeto es una instancia física de la clase.
- 2.** Una clase se define utilizando la palabra clave class. Dentro de la declaración class específicas el código y los datos que componen esa clase.
- 3.** Cada objeto de la clase tiene su propia copia de las variables de instancia de la clase.
- 4.** MiContador = contador;  
contador = new MiContador();
- 5.** double MiMet(int a, int b) { // ...

- 6.** Un método que regresa un valor debe hacerlo a través de la declaración `return`, transmitiendo en el proceso el valor de regreso.
- 7.** Un constructor tiene el mismo nombre que su clase.
- 8.** El operador `new` reserva memoria para un objeto y lo inicializa utilizando el constructor del objeto.
- 9.** El recolector de basura es el mecanismo que recicla objetos que no están siendo utilizados con el fin de liberar el espacio que ocupan en la memoria. Un destructor es un método que es invocado justo antes de que el objeto sea reciclado.
- 10.** Para un método, la palabra clave `this` es una referencia al objeto sobre el cual el método es invocado. Para un constructor, `this` es una referencia al objeto que está siendo construido.

## Capítulo 5: Más tipos de datos y operadores

- 1.** `double[] x = new double[12];`
- 2.** `int[,] nums = new int[4, 5];`
- 3.** `int [][] nums = new int[5][];`
- 4.** `int[] x = { 1, 2, 3, 4, 5 };`
- 5.** El loop `foreach` hace ciclos a través de una colección, obteniendo cada elemento en turno. Su formato general es:

`foreach (tipo nombre-var in colección) declaración;`

- 6.** // Promedia 10 valores double.

```
using System;

class Promedio {
    static void Main() {
        double[] nums = { 1.1, 2.2, 3.3, 4.4, 5.5,
                         6.6, 7.7, 8.8, 9.9, 10.1 };
        double sum = 0;

        for(int i=0; i < nums.Length; i++)
            sum += nums[i];

        Console.WriteLine("Promedio: " + sum / nums.Length);
    }
}
```

- 7.** // Muestra el orden bubble con cadenas de caracteres.

```
using System;

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

```
string[] strs = {
    "esto", "es", "un", "test",
    "de", "un", "orden", "string"
};
int a, b;
string t;
int size;

size = strs.Length; // cantidad de elementos a ordenar

// Muestra el arreglo original.
Console.WriteLine("El arreglo original es:");
for(int i=0; i < size; i++)
    Console.Write(" " + strs[i]);
Console.WriteLine();

// Éste es el orden bubble de las cadenas.
for(a=1; a < size; a++) {
    for(b=size-1; b >= a; b--) {
        if(strs[b-1].CompareTo(strs[b]) > 0) {
            // Cambia el orden de los elementos.
            t = strs[b-1];
            strs[b-1] = strs[b];
            strs[b] = t;
        }
    }
}

// muestra el arreglo ordenado.
Console.WriteLine("El arreglo ordenado es:");
for(int i=0; i < size; i++)
    Console.Write(" " + strs[i]);
Console.WriteLine();
}
```

**8.** El método `IndexOf()` encuentra la primera aparición de la subcadena especificada. `LastIndexOf()` encuentra la última aparición.

**9.** // Un cifrador XOR mejorado.

```
using System;

class Encode {
    static void Main() {
        string msg = "Ésta es una prueba";
        string encmsg = "";
        string decmsg = "";
        string key = "abcdefghijkl";
        int j;
```

```
Console.WriteLine("Mensaje original: ");
Console.WriteLine(msg);

// Codifica el mensaje.
j = 0;
for(int i=0; i < msg.Length; i++) {
    encmsg = encmsg + (char) (msg[i] ^ key[j]);
    j++;
    if(j==8) j = 0;
}

Console.WriteLine("Mensaje codificado: ");
Console.WriteLine(encmsg);

// Decodifica el mensaje.
j = 0;
for(int i=0; i < msg.Length; i++) {
    decmsg = decmsg + (char) (encmsg[i] ^ key[j]);
    j++;
    if(j==8) j = 0;
}

Console.WriteLine("Mensaje decodificado: ");
Console.WriteLine(decmsg);
}
```

**10. No.**

**11. `y = x < 0 ? 10 : 20;`**

**12. Es un operador lógico porque los operandos son del tipo bool.**

## Capítulo 6: Un acercamiento a los métodos y a las clases

**1. No, un miembro privado no puede ser accesado fuera de su clase. Como se explicó, cuando no está presente un especificador de acceso, el acceso por defecto para el miembro de una clase es privado.**

**2. anteceder**

**3. // Una clase de pila para caracteres.**

```
using System;

class Pila {
    char[] pila; // éste es el arreglo que contiene la pila
    int tos;      // parte superior de la pila

    // Construye una pila vacía dado su tamaño.
```

```
public Pila(int tamaño) {
    pila = new char[tamaño]; // reserva memoria para la pila
    tos = 0;
}

// Construye una pila a partir de otra.
public Pila(Pila ob) {
    tos = ob.tos;
    pila = new char[ob.pila.Length];

    // Copia elementos.
    for(int i=0; i < tos; i++)
        pila[i] = ob.pila[i];
}

// Construye una pila con valores iniciales.
public Pila(char[] a) {
    pila = new char[a.Length];

    for(int i = 0; i < a.Length; i++) {
        Push(a[i]);
    }
}

// Empuja los caracteres de la pila.
public void Push(char ch) {
    if(tos == pila.Length) {
        Console.WriteLine(" -- Pila llena.");
        return;
    }

    pila[tos] = ch;
    tos++;
}

// Saca un carácter de la pila.
public char Pop() {
    if(tos == 0) {
        Console.WriteLine(" -- La pila está vacía.");
        return (char) 0;
    }

    tos--;
    return pila[tos];
}
```

```
// Muestra el funcionamiento de la clase pila.
class SDemo {
    static void Main() {
        // construye una pila vacía para 10 elementos.
        Pila stk1 = new Pila(10);

        char[] name = { 'T', 'o', 'm' };

        // Construye una pila a partir del arreglo.
        Pila stk2 = new Pila(name);

        char ch;
        int i;

        // Coloca algunos caracteres en stk1.
        for(i=0; i < 10; i++)
            stk1.Push((char) ('A' + i));

        // Construye una pila a partir de otra.
        Pila stk3 = new Pila(stk1);

        // Muestra las pilas.
        Console.WriteLine("Contenido de stk1: ");
        for(i=0; i < 10; i++) {
            ch = stk1.Pop();
            Console.Write(ch);
        }

        Console.WriteLine("\n");

        Console.WriteLine("Contenido de stk2: ");
        for(i=0; i < 3; i++) {
            ch = stk2.Pop();
            Console.Write(ch);
        }

        Console.WriteLine("\n");

        Console.WriteLine("Contenido de stk3: ");
        for(i=0; i < 10; i++) {
            ch = stk3.Pop();
            Console.Write(ch);
        }
    }
}
```

**Los datos generados por el programa son:**

Contenido de stk1: JIHGFEDCBA

Contenido de stk2: mot

Contenido de stk3: JIHGFEDCBA

**4.** void Swap(Test ob1, Test ob2) {  
    int t;  
  
    t = ob1.a;  
    ob1.a = ob2.a;  
    ob2.a = t;  
}

**5. No.** Los métodos sobrecargados pueden tener diferentes tipos de regreso, pero no tienen injerencia en la resolución sobrecargada. Los métodos sobrecargados *deben* tener listas de parámetros diferentes.

**6.** // Muestra una cadena de caracteres invertida usando recursión.

```
using System;  
  
class Inversa {  
    string str;  
  
    public Inversa(string s) {  
        str = s;  
    }  
  
    public void Invierte(int idx) {  
        if(idx != str.Length-1) Invierte(idx + 1);  
  
        Console.Write(str[idx]);  
    }  
}  
  
class InversaDemo {  
    static void Main() {  
        Inversa s = new Inversa("Esto es una prueba");  
  
        s.Invierte(0);  
    }  
}
```

**7. Las variables compartidas son declaradas como static.**

**8. El modificador ref provoca que un argumento sea transmitido como referencia.** Esto permite que el método modifique el contenido del argumento. Un parámetro ref puede recibir información transmitida en un método. El modificador out es el mismo que ref, excepto que no puede utilizarse para transmitir un valor a un método.

```
9. static void Main()
    static void Main(string[] args)
    static int Main()
    static int Main(string[] args)
```

- 10.** Todos son legales.

## Capítulo 7: Operador sobrecargado, indexadores y propiedades

**1.** public static *tipo-ret* operator *op(tipo-param operando)*
 {
 // operaciones
 }

**El operando debe ser del mismo tipo de la clase por la cual el operando se sobrecarga.**

**2.** Para permitir la mezcla completa del tipo de la clase con un tipo integrado, debes sobrecargar el operador de dos maneras. Una de ellas tiene el tipo de clase como primer operando y el tipo integrado como segundo operando. La segunda manera tiene el tipo integrado como primer operando y el tipo de la clase como segundo operando.

**3.** No, ? no puede ser sobrecargado. No, no puedes cambiar la precedencia de un operador.

**4.** Un indexador proporciona un acceso al objeto de manera semejante a un arreglo.

```
tipo-elemento this[int índice] {
  // El accesador get
  get {
    // Regresa los valores especificados por el índice.
  }
  // El accesador set.
  set {
    // Establece los valores especificados por el índice.
  }
}
```

**5.** El accesador get obtiene el valor especificado por el índice y el accesador set establece el valor especificado por el índice.

**6.** Una propiedad proporciona acceso controlado a un valor.

```
nombre tipo {
  get {
    // Obtiene el código accesador.
  }
  set {
    // Establece el código accesador.
  }
}
```

**7.** No, una propiedad no define una locación de almacenamiento. Maneja acceso a un campo definido por separado. Así, debes declarar un campo que será manejado por la propiedad o utilizar una propiedad autoimplementada, para la cual el compilador proporciona el campo.

**8.** No, una propiedad no puede ser transmitida como parámetro ref o out.

**9.** Una propiedad autoimplementada es aquella para la cual el compilador proporciona automáticamente una variable anónima, llamada campo trasero, para contener el valor. Se indica al especificar solamente get; y set; sin cuerpo.

**10.** // Determina si un conjunto es subconjunto de otro.

```
public static bool operator <(Set ob1, Set ob2) {
    if(ob1.Length > ob2.Length) return false; // ob1 tiene más elementos

    for(int i=0; i < ob1.Length; i++)
        if(ob2.find (ob1[i]) == -1) return false;

    return true;
}
```

// Determina si un conjunto es un superconjunto de otro.

```
public static bool operator >(Set ob1, Set ob2) {
    if(ob1.Length < ob2.Length) return false; // ob1 tiene menos elementos

    for(int i=0; i < ob2.Length; i++)
        if(ob1.find(ob2[i]) == -1) return false;

    return true;
}
```

**11.** // Intersección de conjuntos.

```
public static Set operator &(Set ob1, Set ob2) {
    Set newset = new Set();

    // Añade elementos comunes a los dos conjuntos.
    for(int i=0; i < ob1.Length; i++)
        if(ob2.find(ob1[i]) != -1)
            // Añade el elemento si es miembro de ambos conjuntos.
            newset = newset + ob1[i];

    return newset; // regresa la intersección
}
```

## Capítulo 8: Herencia

**1.** No, una clase base no tiene conocimiento de sus clases derivadas. Sí, una clase derivada tiene acceso a todos los miembros no privados de su clase base.

**2.** // Una subclase de DosDForma para círculos.

```
class Círculo : DosDForma {
    // Construye círculo
    public Círculo(double x) : base(x, "círculo") {}

    // Construye una copia de un objeto.
    public Círculo(Círculo ob) : base(ob) {}

    public override double Area() {
        return (Ancho / 2) * (Ancho / 2) * 3.1416;
    }
}
```

**3.** Para prevenir que una clase derivada tenga acceso a un miembro de la clase base, declara ese miembro como private.

**4.** La palabra clave base **tiene dos formatos**. El primero es utilizado para invocar un constructor de la clase base. El formato general de su uso es

base (lista-parámetros)

El segundo formato de base se utiliza para suscribir un miembro de la clase base. Tiene el siguiente formato general:

base.miembro

**5.** Los constructores son siempre invocados en orden de derivación. Así, cuando se crea un objeto Gamma, el orden es Alfa, Beta, Gamma.

**6.** Cuando se invoca un método virtual a través de una referencia a una clase base, es el tipo del objeto que está siendo referido lo que determina cuál versión del método es invocada.

**7.** Una clase abstracta contiene al menos un método abstracto.

**8.** Para prevenir que una clase se herede, déclarala como sealed.

**9.** Herencia, métodos virtuales y las clases abstractas soportan polimorfismo al permitirte crear una estructura de clase generalizada que pueda ser implementada por una variedad de clases. De esta manera, la clase abstracta define una interfaz consistente, que es compartida por todas las clases implementadas. Esto encarna el concepto de “una interfaz, múltiples métodos”.

**10.** objeto

**11.** Encajonamiento es el proceso de almacenar un tipo de valor en un objeto. El encajonamiento ocurre automáticamente cuando asignas un valor a una referencia object.

**12.** Un miembro protected está disponible para ser utilizado por las clases derivadas, pero en caso contrario es privado para su clase.

## Capítulo 9: Interfaces, estructuras y enumeraciones

- 1.** La interfaz es el mejor ejemplo del principio de la POO “una interfaz, múltiples métodos”.
- 2.** Una interfaz puede ser implementada por una cantidad ilimitada de clases. Una clase puede implementar tantas interfaces como desee.
- 3.** Sí, las interfaces pueden ser heredadas.
- 4.** Sí, una clase debe implementar todos los miembros definidos por una interfaz.
- 5.** No, una interfaz no puede definir un constructor.

```
6. interface IVehículo {  
    int Range();  
  
    double GasNecesaria(int kms);  
  
    int Pasajeros {  
        get;  
        set;  
    }  
  
    int Capacidad {  
        get;  
        set;  
    }  
  
    int Kpl {  
        get;  
        set;  
    }  
}
```

- 7.** // Crea una interfaz de falla leve.  
using System;  
  
// Ésta es la interfaz de falla leve.  
public interface IFallaLeve {  
 // Esto especifica la propiedad length de la interfaz.  
 int Length {  
 get;  
 }  
  
 // Esto especifica la interfaz indexadora.  
 int this[int index] {  
 get;  
 set;  
 }  
}

```
// Ahora, implementa IFallaLeve.
class ArregloFallaLeve : IFallaLeve {
    int[] a; // referencia al arreglo
    int len; // variable trasera para la propiedad length

    public bool ErrFlag; // indica el resultado de la última operación

    // Construye un arreglo dado su tamaño.
    public ArregloFallaLeve(int tamaño) {
        a = new int[tamaño];
        len = tamaño;
    }

    // Una propiedad Length sólo-lectura.
    public int Length { get{ return len; } }

    // Éste es el indexador para ArregloFallaLeve.
    public int this[int index] {
        // Éste es el accesador get.
        get {
            if(ok(index)) {
                ErrFlag = false;
                return a[index];
            } else {
                ErrFlag = true;
                return 0;
            }
        }

        // Éste es el accesador set
        set {
            if(ok(index)) {
                a[index] = value;
                ErrFlag = false;
            }
            else ErrFlag = true;
        }
    }

    // Regresa true si el índice está dentro de los límites.
    private bool ok(int index) {
        if(index >= 0 & index < Length) return true;
        return false;
    }
}
```

```
// Muestra el funcionamiento del arreglo de falla leve mejorado.
class FLMejoradoDemo {
    static void Main() {
        ArregloFallaLeve fs = new ArregloFallaLeve(5);
        int x;

        for(int i=0; i < fs.Length; i++)
            fs[i] = i*10;

        for(int i=0; i < fs.Length; i++) {
            x = fs[i];
            if(x != -1) Console.WriteLine(x + " ");
        }
        Console.WriteLine();
    }
}
```

**8. Una struct define un tipo de valor. Una class define un tipo de referencia.**

**9. enum Planetas {Mercurio, Venus, Tierra, Marte, Júpiter,  
Saturno, Urano, Neptuno };**

## Capítulo 10: Manejo de excepciones

- 1. System.Exception está en la parte superior de la jerarquía de excepciones.**
- 2. try y catch funcionan juntos. Las declaraciones de programas que quieras monitorear para excepciones están contenidas dentro del bloque try. Una excepción es capturada utilizando catch.**
- 3. No hay un bloque try que anteceda la cláusula catch.**
- 4. Si no se captura una excepción, da como resultado una finalización anormal del programa.**
- 5. En el fragmento, una clase base catch antecede una clase catch derivada. Como la clase base catch también capturará todas las clases derivadas, se crea código inalcanzable.**
- 6. Sí, una excepción puede lanzarse de nueva cuenta.**
- 7. Falso. El bloque finally es el código que se ejecuta cuando finaliza un bloque try.**
- 8. He aquí una manera de añadir un controlador de excepciones a la clase Pila:**

```
// Una excepción para errores de pila llena.
using System;

class PilaLlenaExcep : Exception {
    // Crea constructores estándar.
    public PilaLlenaExcep() : base() { }
    public PilaLlenaExcep(string str) : base(str) { }
    public PilaLlenaExcep(string str, Exception inner) :
        base (str, inner) { }
```

```
protected PilaLlenaExcep(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
    base(si, sc) { }
}

// Una excepción para errores de pila vacía.
class ExcepPilaLlena : Exception {
    // Crea constructores estándar.
    public ExcepPilaLlena() : base() { }
    public ExcepPilaLlena(string str) : base(str) { }
    public ExcepPilaLlena(string str, Exception inner) :
        base(str, inner) { }
protected ExcepPilaLlena(
    System.Runtime.Serialization.SerializationInfo si,
    System.Runtime.Serialization.StreamingContext sc) :
    base(si, sc) { }
}

// Una clase pila para caracteres.
class Pila {
    char[] pil; // este arreglo contiene la pila
    int tos;// parte superior de la pila

    // Construye una pila vacía dado su tamaño.
    public Pila(int tamaño) {
        pil = new char[tamaño]; // reserva memoria para la pila
        tos = 0;
    }

    // Construye una pila a partir de otra.
    public Pila(Pila ob) {
        tos = ob.tos;
        pil = new char[ob.pil.Length];

        // Copia elementos.
        for(int i=0; i < tos; i++)
            pil[i] = ob.pil[i];
    }

    // Construye una pila con valores iniciales.
    public Pila(char[] a) {
        pil = new char[a.Length];

        for(int i = 0; i < a.Length; i++) {
            try {
                Push(a[i]);
            }
        }
    }
}
```

```
        catch(PilaLlenaExcep exc) {
            Console.WriteLine(exc);
        }
    }

// Empuja caracteres dentro de la pila.
public void Push(char ch) {
    if(tos==pil.Length)
        throw new PilaLlenaExcep();

    pil[tos] = ch;
    tos++;
}

// Extrae caracteres de la pila.
public char Pop() {
    if(tos==0)
        throw new ExcepPilaLlena();

    tos--;
    return pil[tos];
}
}
```

- 9.** checked y unchecked determinan si un sobreflujo aritmético provoca o no una excepción. Para evitar una excepción, marca el código relacionado como unchecked. Para levantar una excepción de sobreflujo, marca el código relacionado como checked.

- 10.** Todas las excepciones pueden ser capturadas utilizando cualquiera de estos formatos de catch:

```
catch { }

catch(Exception exc) { }

catch(Exception) { }
```

## Capítulo 11: Utilizar E/S

- 1.** Los flujos de bytes son útiles para E/S de archivos, especialmente E/S de archivos binarios, y soportan archivos de acceso aleatorio. Los flujos de caracteres están optimizados para Unicode.

- 2.** La clase en la parte más alta de la jerarquía de flujo es Stream.

- 3.** He aquí una manera de abrir un archivo para entrada de bytes:

```
FileStream fin = new FileStream("miarchivo", FileMode.Open);
```

- 4.** He aquí una manera de abrir un archivo para lectura de caracteres:

```
StreamReader frdr_in = new StreamReader(new FileStream("miarchivo",
    FileMode.Open));
```

- 5.** Seek( ) establece la posición actual del archivo.
- 6.** E/S Binaria para los tipos integrados de C# es soportada por BinaryReader y BinaryWriter.
- 7.** E/S es redireccionada al invocar SetIn( ), SetOut( ) y SetError( ).
- 8.** Una cadena numérica puede ser convertida en su representación interna utilizando el método Parse( ) definido en las estructuras alias de .NET.
- 9.** /\* Copia un archivo de texto, sustituyendo guiones por espacios.

Esta versión usa flujos de byte.

Para utilizar este programa, especifica el nombre del archivo fuente y el archivo destino.  
Por ejemplo:

```
Guión fuente destino
*/
using System;
using System.IO;

class Guión {
    static void Main(string[] args) {
        int i;
        StreamReader fin;
        StreamWriter fout;

        if(args.Length != 2) {
            Console.WriteLine("Uso: Guión De Hacia");
            return;
        }

        // Abre el archivo de entrada.
        try {
            fin = new StreamReader(args[0], FileMode.Open);
        } catch(IOException exc) {
            Console.WriteLine(exc.Message);
            return;
        }

        // Abre el archivo de salida.
        try {
            fout = new StreamWriter(args[1], FileMode.Create);
        } catch(IOException exc) {
            Console.WriteLine(exc.Message);
            fin.Close();
            return;
        }
    }
}
```

```
// Copia el archivo.  
try {  
    do {  
        i = fin.ReadByte();  
        if((char)i == '-') i = '-';  
        if(i != -1) fout.WriteByte((byte) i);  
    } while(i != -1);  
} catch(IOException exc) {  
    Console.WriteLine(exc.Message);  
}  
  
fin.Close();  
fout.Close();  
}  
}
```

**10.** /\* Copia un archivo de texto, sustituyendo guiones por espacios.

Esta versión usa flujos de caracteres.

Para usar este programa, especifica el nombre  
del archivo fuente y el archivo destino.  
Por ejemplo:

```
Hyphen source target  
*/  
  
Using System;  
Using System.IO;  
  
Class Hyphen {  
    Static void Main(string[] args) {  
        int i;  
        StreamReader fin;  
        StreamWriter fout;  
  
        if(args.Length != 2) {  
            Console.WriteLine("Usage:Hyphen From To");  
            return;  
        }  
  
        // Open the input file.  
        try {  
            fin = new StreamReader(args[0]);  
        } Catch(IOException exc) {  
            Console.WriteLine(exc.Message);  
            return;  
        }
```

```

// Copia el archivo.
try {
    fout = new StreamWriter(args[1]);
} catch(IOException exc) {
    Console.WriteLine(exc.Message);
    fin.Close();
    Return;
}

// Copia el archivo.
try {
    do {
        i = fin.Read();
        if((char)i == ' ') i = '-';
        if(i != -1) fout.Write((char)i);
    } while(i != -1);
} catch(IOException exc) {
    Console.WriteLine(exc.Message);
}

fin.Close();
fout.Close();
}
}

```

## Capítulo 12: Delegados, eventos y nomenclaturas

- 1.** delegate double Filter(int i)
- 2.** Una transmisión múltiple es creada añadiendo métodos a una cadena de delegados utilizando el operador `+=`. Un método puede ser eliminado utilizando `-=`.
- 3.** Un método anónimo es un bloque de código que es transmitido a un constructor de delegado.
- 4.** Sí. Sí.
- 5.** Un evento requiere el uso de un delegado.
- 6.** event
- 7.** Sí, los eventos pueden ser objeto de transmisión múltiple.
- 8.** Un evento siempre es enviado a una instancia específica.
- 9.** Una nomenclatura define una región declarativa, que previene colisiones entre nombres.
- 10.** using alias = nombre;
- 11.** namespace X.Y {
 // ...
}

## Capítulo 13: Genéricos

**1.** Verdadero.

**2.** Un parámetro de tipo es un marcador de posición para un tipo auténtico. El tipo auténtico será especificado cuando un objeto de clase genérica o un delegado es inicializado, o cuando se invoca un método genérico.

**3.** Un argumento de tipo es un tipo de dato auténtico que es transmitido a un parámetro de tipo.

**4.** Cuando se opera a través de referencias object, el compilador no capturará errores de inconsistencia de tipo entre el tipo de dato proporcionado y el tipo de dato requerido. Esto provoca los errores que ocurren en tiempo de ejecución.

**5.** Una clase base limitada en un parámetro de tipo requiere que cualquier argumento de tipo especificado para ese parámetro sea derivado de o idéntico a la clase base especificada. La limitación tipo-valor requiere que un argumento de tipo sea de un tipo valor, no de un tipo referencia.

**6.** Sí.

**7.** Para obtener el valor por defecto de un tipo, utiliza el operador default.

**8.** La lista de parámetros de tipo para un método genérico sigue inmediatamente al nombre del método. Así, va entre el nombre del método y el paréntesis de apertura de la lista de parámetros del método.

**9.** Sí.

**10.** He aquí un acercamiento. Utiliza propiedades autoimplementadas para X y Y.

```
public interface IDosDCoord<T> {
    T X { get; set; }
    T Y { get; set; }
}

class XYCoord<T> : IDosDCoord<T> {
    public XYCoord(T x, T y) {
        X = x;
        Y = y;
    }

    public T X { get; set; }
    public T Y { get; set; }
}
```

## Capítulo 14: Introducción a LINQ

**1.** LINQ quiere decir Lenguaje de Query Integrado (*Language Integrated Query*). Se usa para recuperar información de una fuente de datos.

**2.** IEnumerable en cualquiera de sus dos formatos, genérico o no genérico.

- 3.** Las palabras clave de query son from, group, where, join, let, orderby y select.
- 4.** var todoData = from info in miListaDatos  
select info;
- 5.** var altoData = from info in miListaDatos  
select info.Alto;
- 6.** where  
  
var todoData = from info in miListaDatos  
where info.Alto < 100  
select info.Alto;
- 7.** orderby  
  
var todoData = from info in miListaDatos  
where info.Alto < 100  
orderby info.Alto descending  
select info.Alto;
- 8.** group  
group *variable-rango* by *clave*
- 9.** join  
from *rango.varA* in *fuente-datosA*  
join *rango.varB* in *fuente-datosB*  
on *rango-varA.propiedad* equals *rango-varB.propiedad*
- 10.** into
- 11.** let
- 12.** new { Autor = "Knuth", Título = "El Arte de la Programación de Cómputo" }
- 13.** =>
- 14.** True
- 15.** Where()
- 16.** Una extensión de método añade funcionalidad a una clase, pero sin utilizar la herencia.  
El primer parámetro define el tipo que será extendido. Debe ser modificado por this. Al primer parámetro es transmitido automáticamente el objeto sobre el cual se invoca la extensión del método.
- 17.** Las extensiones de método son importantes para LINQ porque los métodos de query son implementados como extensiones de método para IEnumerable<T>.

## Capítulo 15: El preprocesador, RTTI, tipos anulables y otros temas avanzados

1. Las directivas de compilación condicional son #if, #elif, #else y #endif.
2. #elif significa “else if.” Te permite crear una cadena if-else-if para opciones de compilación múltiple.

- 3.** Para obtener un objeto System.Type en tiempo de ejecución, utiliza el operador `typeof`.
- 4.** La palabra clave `is` determina si un objeto es de un tipo especificado.
- 5.** El operador `as` convierte un tipo en otro si la conversión es un encajonamiento, desencajonamiento, identidad o referencia legal. Su beneficio consiste en que no lanza una excepción en caso de que la conversión no sea permitida.
- 6.** `int? cuenta = null;`
- 7.** `True`.
- 8.** El operador `??` es llamado el operador nulo coalescente. Regresa el valor de una instancia anulable si esa instancia contiene un valor. En caso contrario, regresa el valor por defecto especificado.
- 9.** El código inseguro es el que se ejecuta fuera del contexto controlado.
- 10.** `double* ptr;`
- 11.** Los atributos están encerrados entre corchetes, como se muestra aquí:  
[atributo]
- 12.** Existen operadores de conversión implícitos y explícitos. Sus formatos generales se muestran a continuación:  

```
public static operator implicit tipo-objetivo(tipo-fuente v) { return valor; }  
public static operator explicit tipo-objetivo(tipo-fuente v) { return valor; }
```
- 13.** Una colección es un conjunto de objetos. Las colecciones genéricas están en la nomenclatura `System.Collections.Generic`.
- 14.** `ICollection<T>`
- 15.** La palabra clave `partial` es usada en ambos casos.

# Índice

## Símbolos

---

- operador de resta, 23, 61-62  
y delegados, 437  
--, 31, 61, 62-64, 250  
!, 64, 65, 566  
!=, 28, 64, 261, 497, 566  
y cadena de caracteres, 179  
#, 49, 554  
% , 61, 62  
&  
AND bitwise, 182-184  
AND lógico, 64, 65, 68-69, 566  
operador de puntero, 586

.NET Framework, 2, 6-8  
biblioteca de clases, 7, 8, 19, 37  
Colecciones API, 578, 580  
sistema E/S, 392  
y C++, 8  
y eventos, 449  
/, 23, 61-62  
/\*\*/, 18  
//, 19  
/ unsafe, opción de compilador, 567  
:: calificador alias de nomenclatura, 458  
; (punto y coma), 20, 32-33  
?  
operador, 192-194  
para declaraciones de tipo anulable, 564

|  
 OR bitwise, 182, 183, 184-185  
 OR lógico, 64, 65, 68-69  
 || (circuito corto OR), 64, 66-67, 68-69  
 ~  
 NOT bitwise, 182, 183, 187  
 usado por destructor, 148  
 +  
 operador de suma, 23, 61-62  
 uso con WriteLine( ), 23  
 y concatenación de cadenas, 180  
 y delegados, 437  
 ++, 31, 61, 62-64, 250  
 +=  
 y delegados, 437  
 y eventos, 445  
 <  
 operador de redirección E/S, 410-411  
 operador relacional, 28, 64, 261, 566  
 <<, 182, 187-189  
 <=, 28, 64, 261, 566  
 < >, 466, 467  
 =, 23, 69  
 -=  
 y delegados, 437  
 y eventos, 445  
 == (operador relacional), 28, 64, 261, 328, 497,  
     566  
     y cadena de caracteres, 179  
=> operador lambda, 535, 545  
>  
 operador de redirección E/S, 410-411  
 operador relacional, 28, 64, 261, 566  
->, 568-569  
>=, 28, 64, 261, 566  
>>, 182, 187-189

## A

---

Abs( ), 223  
 Accesadores  
     indexador, 263-266, 267  
     propiedad, 270, 272, 274-277  
 Add( ), 579  
 ADO.NET bases de datos y LINQ,  
     507  
 Advertencia #pragma, opción, 560-561  
 Alcances, 58-61  
     y declaración de nomenclaturas,  
         450  
 All( ) método de extensión, 539-540  
 Ambiente Windows, 5, 7

AND bitwise (&), 182, 184  
 circuito corto (&&&), 64, 66-67, 68-69  
 lógico (&), 64, 65, 68-69, 566  
 Anulables, tipos, 41, 483, 563-566  
     especificar, 563, 564  
     y el operador ??, 565-566  
     y el operador lógico y relacional, 566  
 Any( ) método de extensión, 539-540  
 ApplicationException, 378  
 archivo exe, 17  
 Archivo(s)  
     E/S. Ver E/S, archivo  
     indicador de posición/puntero, 418  
 ArgumentException, 401, 407, 409, 410, 414, 415  
 ArgumentNullException, 401, 407, 409, 410, 414, 422  
 Argumento(s), 20, 128, 134  
     cantidad variable de, usar, 213-216  
     línea de comandos, 231-233  
     tipo. Ver Argumentos de tipo  
     transmitir, 206-208  
 Argumentos de tipo, 467, 468  
     para invocar un método genérico, usando explícitos,  
         491-492  
     y limitaciones, 473-485  
 ArgumentOutOfRangeException, 401  
 ArrayTypeMismatchException, 378  
 Arreglo(s), 154-174  
     almacenar, 159-160, 239-242  
     de cadenas, 180-181  
     dentados, 163-165, 168  
     desbordamiento, prevenir, 200-203, 272-273, 275-277  
     dinámicos, usar List<T> para crear un, 580-583  
     implementados en C# como objetos, 154, 155, 167  
     inicializar, 157-158, 162-163  
     límites, 158  
     multidimensionales, 161-163  
     parámetros param, 213-216  
     propiedad Length de, 167-169, 203, 272-273, 275-277  
     tamaño fijo, 571-572  
     tipo implícito, 169-170  
     unidimensionales, 154-160  
     variables de referencia, asignación, 165-167  
     y la interfaz IEnumerable<T>, 508, 509, 512, 533, 536  
     ascendente palabras clave contextuales, 512, 515  
 ASCII conjunto de caracteres, 46, 392-393  
     mayúsculas y minúsculas, 183, 185  
 Asignaciones  
     conversión de tipo en, 70-73  
     variables de referencia y, 127-127  
 ASP.NET, 561  
 Atributo condicional integrado, 572-573  
 Atributos, 572-574  
 Average( ), método de extensión, 539-541

**B**

base, palabra clave  
 para accesar un miembro oculto de clase base, 298, 303-305  
 para invocar un constructor de clase base, 298-302, 311  
 Biblioteca de clases .Net Framework, 7, 8, 9, 37  
 BinaryReader, clase, 396, 414, 415-417  
 métodos de entrada, tabla de los usados comúnmente, 416  
 BinaryWriter, clase, 396, 414, 416-417  
 métodos de salida, tabla de los usados comúnmente, 415  
 Bitwise, operadores, 182-192  
 Bloques de código, 20, 31-32, 58  
 bool, tipo simple, 41-47  
 valor por defecto, 143  
 y operadores lógicos, 65  
 y operadores relacionales, 64  
 break, declaración, 82, 87, 88, 89-90, 106-108  
 Bubble, orden, 159-160  
 Buffer, tamaño fijo, 571-572  
 BufferedStream, clase, 394  
 Byte, estructura, 421  
 byte, tipo simple, 41, 42, 43, 52  
 Bytecode, 4, 7

**C**

C, historia de, 3  
 y Java, 4-5, 6  
 C#  
 biblioteca de clases, 7, 8, 19, 37  
 como lenguaje orientado a componentes, 6  
 como un programa estricto de tipos, 40, 313  
 compilación de programa, 7, 12-17  
 historia de, 2-3, 5-6  
 palabras clave. *Ver* Palabras clave, C#.br/>
 y .NET Framework, 2, 6-8  
 y Java, 3, 6  
 C++  
 cuándo utilizar en vez de C#, 567  
 historia de, 3  
 y .NET Framework, 8  
 y Java, 4-5, 6  
 Cadenas de caracteres  
 arreglos de, 180-181  
 como objetos, 177  
 construir, 177  
 indexar, 178  
 inmutabilidad de la clase string, 181-182  
 literales, 53-55, 177

numéricas, convertir, 420-424  
 propiedad Length de, 178  
 campo sólo-lectura, 587  
 Campo trasero, 274, 277  
 CanRead, propiedad, 395  
 CanSeek, propiedad, 395  
 CanTimeOut, 395  
 CanWrite, propiedad, 395  
 Capacity, propiedad, 581  
 Caracter(es), 46  
 codificación, 410  
 desde el teclado, entrada, 82-83  
 literales, 51, 55  
 secuencias de escape, 52-53  
 case, declaración, 87, 89-92  
 Cast, 72-73, 74, 75-76, 465, 468  
 Cast vs. el operador as, 562  
 Cast y operadores de conversión explícitos, 574, 577, 578  
 catch, cláusula, 363-366  
 múltiples, uso, 369  
 para capturar todas las excepciones, 370  
 y capturar excepciones de clases derivadas, 380-382  
 y relanzamiento de excepciones, 373-374  
 char, tipo simple, 23, 41-42, 46  
 checked, palabra clave, 386-388  
 checksum #pragma, opción, 561  
 Clase genérica  
 ejemplo con dos parámetros de tipo, 471-472  
 ejemplo con un parámetro de tipo, 465-468  
 formato general de, 472  
 Clase(s), 19, 120-125  
 abstracta, 323-326, 334  
 base, definición de, 288  
 biblioteca, .NET, 7, 8, 19, 37  
 bien diseñada, 122, 137  
 como tipo de referencia, 126, 355, 357  
 constructor. *Ver* Constructor(es)  
 definición de, 7, 10, 120  
 derivadas, definición, 288  
 e interfaces, 334-339  
 estática, 239  
 formato general de, 121-122, 291  
 genérica. *Ver* Clase genérica  
 miembro. *Ver* Miembro, clase  
 nombre y nombre de archivo de programa, 18  
 parcial, 589-590  
 sellada, 326-327  
 Clasificación jerárquica, 10-11  
 Clasificación jerárquica y herencia, 288  
 class, palabra clave, 19, 121, 473, 481  
 Clear( ), 579

- Close( ), 394, 396, 402, 405, 414, 415  
CLR. Ver Lenguaje Común en Tiempo de Ejecución (CLR)  
Código  
    bloques, 20, 31-32, 58  
    controlado, 8, 567  
    controlado vs. sin control, 8, 567  
    inalcanzable, 133  
    inseguro, 567-572  
    sección, crítica, 586-587  
    sin control, 8, 567  
Colas, orden en, 170, 341  
    circular, ejemplo de programa para, 343-344  
    dinámica circular, ejemplo de programa para, 344-345  
    genéricas, programa ejemplo para, 498-503  
    tamaño fijo, programa para, 170-174, 203-205, 228-231, 342-343, 382-385  
    usar Queue<T> para crear una, 583-586  
Colección, 174  
    clases, 550, 578-586  
Colecciones API, 578, 580  
Comentarios, 18, 19, 20  
CompareTo( ), 178, 179, 424, 498  
Complemento Two, 42  
Compilación condicional, 555, 572  
Compilador  
    JIT, 7, 17  
    línea de comando C#, 12, 16-17  
Compile( ), 538  
Complemento de uno (NOT unitario), operador (-), 182, 183, 187  
Componentes y C#, software, 6  
Console, clase, 20, 393, 397, 398, 399, 411  
Console.Error, 393, 397, 398-399  
    redireccionar, 411  
Console.In, 393, 397-398  
    redireccionar, 410-411  
Console.Out, 393, 397, 398-399  
    redireccionar, 410-412  
Console.Read( ), 82  
ConsoleKey, enumeración, 400  
ConsoleKeyInfo, estructura, 400  
ConsoleModifiers, enumeración, 400  
const, modificador, 589  
Constructor(es), 143-146  
    base para invocar una clase base, uso, 298-302, 311  
    en la jerarquía de clase, orden de invocación, 311-312  
    estáticos, 238-239  
    sobrecarga, 224-231  
    y herencia, 296-302, 314-315  
    y tipos anónimos, 528  
Contains( )  
    ICollection<T>, método, 579  
    método de extensión, 539-540  
continue, declaración, 82, 108-109  
Controlador de excepciones, 362-389, 404  
    bloque, formato general, 363  
    cuando utilizar, 389  
    Ver también try, bloque(s)  
    y crear excepciones de clases derivadas, 363, 378-385, 389  
y excepciones no atrapadas, 366-369  
Conversión de tipos  
    ampliar, 71  
    en expresiones, 73, 76  
    estrechar, 72  
    explícita, 72-73  
    implícita, 70-72, 220-222  
    operadores de conversión para, uso, 574-578  
    tiempo de ejecución, usar para, 562  
Copy( ), 178  
CopyTo( ), 579  
Cpunt() método de extensión, 523, 539, 541-542  
csc.exe compilador de línea de comando, 12, 16-17

---

**D**

---

- Datos  
    estructura, 170  
    motor de, 170  
Datos de entrada en consola, 82-83  
Declaración por defecto, 87-88, 89, 90  
    formato de operación, 485-487  
    uso con #line, 560  
Declaraciones, 20, 32-33  
    control, 27, 82  
Declaraciones de reiteración, 82  
Delegados, 432-439  
    declarar, 433  
    genéricos, 492-494  
    instanciar, 435  
    y eventos, 439, 443-444  
    y expresiones lambda, 440, 545-546  
    y métodos anónimos, 440-443  
    y transmisión múltiple, 437-439  
delegate, palabra clave, 433, 440, 441  
Dequeue( ), 583, 584  
descending, palabra clave contextual, 512, 515  
Desencajonar, 329-330  
Destructores, 148  
Dictionary<TK, TV>, colección, 580  
Directiva #define, 555, 559, 572  
Directiva #elif, 555, 557-558  
Directiva #else, 557-558  
Directiva #endif, 555, 556, 557, 558  
Directiva #endregion, 560

Directiva #error, 559  
 Directiva #if, 555-556, 557, 558, 572  
 Directiva #line, 560  
 Directiva #pragma, 560-561  
 Directiva #region, 560  
 Directiva #undef, 559  
 Directiva #warning, 559  
 Directivas de procesador, 554-561  
 DirectoryNotFoundException, 401, 410  
 Dispose(), 402, 588, 589  
 DivideByZeroException, 363, 367, 368, 372, 378  
 Dot, operador, 123, 130, 236, 358  
 Double, estructura, 421  
 double tipo simple, 23, 24-26, 41, 44, 52  
     literal, 52  
 do-while, loop, 82, 102-103

**E**

E/S, 392-430  
     acceso aleatorio de archivos, 418-420  
     consola, 397-400  
     fluxos. *Ver* Flujo(s)  
     redireccionar, 399, 410-412  
     archivo  
         acceso aleatorio, 418-420  
         basado en caracteres, 400, 406-410  
         orientado a bytes, 400-406  
         y datos binarios, 414-418  
 Ecma estándar C++/CLI, 8  
 else, 83-87  
 Encajonado, 329-330  
 Encapsulación, 9-10, 19, 59, 137, 198  
 EndsWith(), 514  
 Enqueue(), 583, 584  
 Ensamblaje, 199, 586  
 Entero(s), 26, 41-43  
     literales, 51, 52  
     promoción, 75  
 enum, palabra clave, 358  
 Enumerable, clase, 534, 539, 550  
 Enumeraciones, 41, 334, 357-360  
 equals, 512, 525, 527  
 Equals(), 261, 327, 328, 424  
 Errores  
     de sintaxis, 20-21  
     en tiempo de ejecución, 362, 470-471  
 Especificación de Lenguaje Común, 8  
 especificador de acceso  
     interno, 199, 586  
     privado, 199-205  
     privado y herencia, 291-294, 295

Especificadores de acceso, 121, 199-205, 586  
     con un accesador, uso, 274-277

Estilo de indentación, 33

Estructura  
     decimal, 421  
     única, 421

Estructura(s), 334, 335-337  
     como tipos de valor, 41, 334, 356, 357  
     genéricas, 487-488  
     .NET numéricas, 420-421, 424  
     parciales, 589, 590  
     valor por defecto de una, 485  
     y buffers de tamaño fijo, 571-572

Etiqueta, 110-111

event, palabra clave, 443

EventArgs, 449

EventHandler, 449

Eventos, 432, 443-449

multitransmisión, 445-448  
     y expresiones lambda, 547-550  
     y genéricos, 472  
     y métodos anónimos, 443, 448-449

Excepción, clase, 362-363, 376-378, 380  
     constructores, 377, 379, 380, 381

Excepciones, 377

integraduras estándar, 362, 363, 378  
     internas, 377

explicit, palabra clave, 574

Expresión(es)

árbol, 538  
     conversión de tipo en, 73-76  
     espacios y paréntesis, 77  
     símbolo, 555-556

Expresiones Lambda, 6, 449, 535-536

como controladores de eventos, uso, 547-550  
     declaración, 535, 544-545, 546-547  
     expresiones, 535-536, 544, 545-546  
     vs. métodos anónimos, 544, 547  
     y árboles de expresión, 538  
     y delegados, 440, 544, 545-546  
     y métodos query para crear queries, uso,  
         536-538

Expression, clase, 538

extern, 592

**F**

false, 47, 64, 261

FileAccess, enumeración, 402

FileMode, enumeración, 400-401

FileNotFoundException, 401, 410

FileStream, clase, 394, 400-406, 414, 415

Finalize( ), 327  
finally, bloque, 363, 374-376  
Firma de un método, 224  
First( ), método de extensión, 539-540  
fixed, palabra clave, uso de, 570-572  
float, tipo simple, 24, 26, 41, 44  
    literal, 52  
Flujo(s)  
    binario, 396  
    byte, 392-393, 400, 406  
    caracteres, 393, 406  
    clases, byte, 394  
    definición de, 392  
    envolver clases, carácter, 394-396  
    memoria base, 430  
    predefinidas, 393  
    redirección estándar, 399, 410-412  
Flush( ), 394, 396, 405, 414  
for, loop, 30-31, 82, 94-100, 102  
    variaciones, 96-100  
foreach, loop, 82, 174-177  
    para ejecutar un query, 508, 510, 512, 541  
FormatException, 422  
Formato especificador de C, 49  
Frank, Ed, 4  
from, cláusula, 509, 512, 513  
    anidado, 523-524  
Func<T, Result> tipo delegado, 534  
Func<Tinner, TKey> tipo, 535  
Func<TOuter, Tinner, TResult> tipo, 535  
Func<TOuter, TKey> tipo, 535  
Function, 10

## G

---

Genéricos, 464  
    prevenir errores en tiempos de ejecución con, 470-471  
    y seguridad de tipos, 464, 465, 468-471, 476, 490,  
        494  
get, accesador  
    para indexador, 263-266, 267, 349  
    para la propiedad autoimplementada, 274  
    para propiedad, 270, 272, 348  
    utilizar modificadores de acceso con, 274-275  
GetEnumerator( ), 579  
GetHashCode( ), 261, 327, 328  
GetType, 327  
Gosling, James, 4  
goto, 82, 108, 109-111  
group, cláusula, 509, 512, 513, 519-521  
    into, usar con una, 521-523  
GroupBy( ), 534, 537

## H

---

Hash, código, 328  
HasValue, propiedad, 564, 565  
Hejlsberg, Anders, 5  
Herencia, 9, 10-11, 288-331  
    acceso a miembros y, 291-294  
    bases de, 288-291  
    e interfaces, 351-352  
    multinivel, 308-311  
    nombres ocultos y, 302-305, 352  
    sellars para prevenir, uso de, 326-327  
    vs. métodos de extensión, 542  
    y constructores, 296-302, 311-312, 314-315  
hidden, 560  
Hilos, múltiples, 586  
Hoare, C.A.R., 239

## I

---

ICollection<T> interfaz, 579, 580  
IComparable, interfaz, 498  
Identificación de tipo en tiempo de ejecución, 561-563  
identificador global predefinido, 458  
Identificadores, 35-36  
IDictionary<TK, TV> interfaz, 580  
IDisposable, interfaz, 589  
IEnumerable, interfaz, 508, 509, 511-512, 523, 534, 536, 539,  
    579, 580  
    y arreglos, 508, 509, 512, 524, 536  
IEnumerable<IGrouping< TKey, TElement >>, interfaz, 520  
IEnumerable<T>, interfaz, 508, 509, 511-512, 523, 534, 536,  
    539, 579, 580  
    y arreglos, 508, 509, 512, 524, 536  
if, declaración, 27-30, 82, 83-87  
    anidado, 85  
    y valores bool, 47, 82  
if-else-if, escalera, 86-87  
    vs. declaración switch, 91  
IGrouping< TKey, TElement > interfaz, 520  
IList<T> interfaz, 580  
implicit, palabra clave, 574  
in, palabra clave contextual, 512  
Indexadores, 262-269  
    interfaz, 349-351  
    multidimensionales, 267-269  
    restricciones, 269  
    sobrecarga, 269  
    sólo-lectura o sólo-escritura, crear, 266  
    unidimensionales, 262-267  
virtual, 318  
y genéricos, 472

IndexOf( ), 178, 580  
 IndexOutOfRangeException, 158, 364, 367, 378  
 Inicializador de proyección, 530  
 InnerException, propiedad, 377  
 Insert( ), 580  
 Instancia de una clase, 120  
*Ver también* objeto(s)  
 Int 16, estructura, 421  
 int, tipo simple, 23, 24, 25, 41, 42, 43, 52  
 int32, estructura, 421  
 Int64, estructura, 421  
 interface, palabra clave, 334, 335  
 Interfaces, 334-335  
     formato general de, 335  
     genéricas, 494-497  
     implementación, 335-339  
     indexadores, 349-351  
     miembro, implementación explícita de una, 353-355  
     parcial, 589  
     propiedades, 347-349  
     variables de referencia, 339-341  
     y herencia, 351-352  
 Internet y portabilidad y seguridad, 4, 5  
 Interoperabilidad de lenguajes mezclados, 5  
 into, cláusula, 512  
     para crear una continuidad, 521-523  
     para crear una unión de grupos, 528, 531-534  
 InvalidCastException, 378  
 InvalidOperationException, 583  
 Invocación por referencia  
     utilizar ref para crear una, 209-210  
     vs. Invocación por valor, 206-208  
 IOException, 393, 395, 397, 401, 403, 404, 405, 408, 410, 415, 418  
 IQueryable<T>, interfaz, 534  
 is, operador, 561-562  
 IsReadOnly, propiedad, 579

**J**

Java  
     características faltantes en, 5  
     historia de, 4-5  
     y C#, 3, 6  
 Java Virtual Machine (JVM), 4  
 JIT, compilador, 7, 17  
 Join( ), 534, 535, 538  
 join, cláusula, 512, 525-528  
     formato general de, 525  
     usar into con, 528, 531-534  
     y tipos anónimos, 528, 529-539  
 Jump, declaración, 82

**K**

Key, propiedad, 400, 520  
 KeyChar, propiedad, 400

**L**

Last( ) método de extensión, 539-540  
 LastIndexOf( ), 178, 521  
 Lenguaje Común en Tiempo de Ejecución (CLR), 7-8, 17, 567  
 Lenguaje Intermediario de Microsoft (MSIL), 7-8, 17  
 let, cláusula, 512, 523-525  
 LINQ (Lenguaje de Query Integrado), 6, 24, 58, 169, 170, 506-524, 550  
     fundamentos, 507-513  
*Ver también* query  
     y archivos XML, 507  
     y conjuntos de datos ADO .NET, 507  
     y expresiones lambda, 535-536  
     y SQL, 507-513  
     y tipos anónimos, 528-531  
 List<T>, colección, 580-583  
 Literales, 51-55  
 Literales hexadecimales, 52  
 Literales octales, 53  
 Length, propiedad  
     de arreglos, 167-169, 203, 272-273, 275-277  
     de cadenas, 178  
     de flujos, 393, 395  
 lock, 586-587  
 long, tipo simple, 41, 42-43  
 long, tipo simple, literales, 52  
 Loops

    anidados, 107-108, 115  
     break y, 106-108  
     criterios para selección, 101  
     do-while, 82, 102-103  
     for. *Ver* for, loop.  
     foreach. *Ver* foreach, loop  
     infinitos, 98, 107  
     sin cuerpo, 98-99  
     while, 82, 100-101

**M**

Main( ), 19-20, 122, 127, 236  
     regresar valor de, 231  
     y argumentos de línea de comandos, 231-233  
 Math, clase, 44  
 Math.Pow( ), 77  
 Math.Sqrt( ), 44, 131

Max( ), método de extensión, 539-540  
MemberwiseClone( ), 327  
MemoryStream, clase, 394, 430  
Mensaje propiedad de Exception, 376-377  
Metadatos, 7-8  
Método(s), 10, 19, 127-137  
    abstractos, 322-326  
    alcance definido por, 59  
    anónimos, 440-443, 448-449, 535, 544, 547  
    base para accesar ocultos, usar, 303, 304-305  
    condicional, 572-573  
    conversión de grupos, 435  
    delegados y, 432-439  
    e interfaces, 334, 335  
    enviar, dinámicos, 316  
    estáticos, 236-238, 323  
    extensión. *Ver* Métodos extendidos  
    firma, 224  
    formato general de, 128  
    genéricos. *Ver* Métodos genéricos  
    implementación explícita de interfaz, 353-355  
    invocar, 130  
    operador, 246-247, 254-259  
    parcial, 589, 590-591  
    query. *Ver* Query, métodos  
    recursivos, 234-235  
    regresar más de un valor de, 209, 211-212  
    regresar objetos de, 216-218  
    regresar valor de, 130, 131-133  
    sobrecarga, 218-224  
    sobrecarga. *Ver* Sobrecarga, métodos  
    transmitir referencia de objetos a, 205-208  
    y parámetros, 128, 134, 137  
Métodos dinámicos, envío, 316  
Métodos extendidos, 6, 239, 542-544, 550  
    correspondientes a palabras clave query, 534-535  
    definición de, 534  
    formato general de, 543  
    relacionadas con el query, 539-541  
    vs. mecanismo de herencia, 542  
    y colecciones, 579  
Métodos genéricos, 488-492  
    formato general de, 490  
    y limitaciones, 491  
    y tipos de argumento explícitos, 491-492  
Métodos virtuales, 315-322  
Métodos virtuales y sellado, 326-327  
Microsoft, 5  
Miembro, clase, 10, 120  
    acceso y herencia, 291-296  
    controlar acceso a, 121, 198-205  
    operador de punto para accesar, 123  
Min() método de extensión, 539-540

modificador de tipo abstracto, 315, 323, 326  
Modificadores, propiedad, 400  
Modulus, operador (%), 61, 62  
Multitarea, 586  
Multitransmisión  
    delegados y, 437-439  
    y eventos, 445-448

---

**N**

namespace, palabra clave, 450  
Naughton, Patrick, 4  
new, 126, 144, 146-147, 155, 157-158, 170, 435  
    ocultar un miembro de clase base, 302-303, 305, 352  
    usando un tipo anónimo, crear, 528, 530  
    y estructuras, 356-357  
new( ) limitaciones de constructor, 473, 479-481, 484  
Nomenclatura(s), 19, 21, 432, 450-458  
    anidadas, 456-457  
    calificador de alias (::), 458  
    característica aditiva de, 455-456  
    crear un alias para, 454-455  
    declarar, 450-452  
    global, 450, 452, 458  
NOT, operador  
    unitario bitwise (-), 182, 183, 187  
    unitario lógico (!), 64, 65  
NotSupportedException, 393, 401, 403, 404, 418, 579, 580  
Null, operador coalescente (??), 565-566  
null, valor, 483, 485, 563-566  
Nullable<T>, 563, 564  
Números negativos, representación de, 188

---

**O**

Oak, 4  
ObjectDisposedException, 403, 404  
Objeto(s), 10, 120, 124  
    crear, 122, 126  
    referencias a métodos, transmitir, 205-208  
    regresar, 216-218  
objeto, clase, 327-330, 465, 468  
    métodos definidos por, 327  
    tipos referencia y valor, 329-330  
Objetos, inicialización  
    con otros objetos, 225-226  
    con un constructor, 143-146  
    y tipos anónimos, 528-529  
    y tipos con nombre, 531

- Obsolete, atributo integrado, 572, 573-574  
 Ocultar nombres  
     y bloques de código, 61  
     y herencia, 302-305, 352  
 on, 512, 525  
 Operador as, 562  
 Operador de decremento (–), 31, 61, 62-64, 250  
 Operador de incremento (++), 31, 61, 62-64, 250  
 Operador sobrecargado, 246-262  
     binario, 247-250  
     consejos y restricciones, 261-262  
     relacionales, 259-261  
     unitario, 247, 250-254  
     y sobrecargar un método de operador, 254-259  
     y verdadero y falso, 261  
 Operador(es)  
     ? ternario, 192-194  
     aritmético, 23, 61-64  
     asignar, 23, 69  
     bitwise, 182-192  
     conversión. *Ver* Conversión, operaciones  
     lambda (=>), 535, 545  
     lógicos, 64-69, 566  
     nulo coalesceante (??), 565-566  
     paréntesis y, 77  
     precedencia, tabla de, 74  
     relacionales, 28, 64-65, 566  
     y genéricos, 472  
 Operador(es) de asignación  
     =, 23, 69  
     aritméticos y lógicos (op=) compuestos, 69-70, 261-262  
     compuestos bitwise, 189  
     taquigráfico, 70  
     y sobrecarga de operadores, 261, 262  
 Operadores aritméticos, 23, 61-64  
 Operadores de cambio, bitwise, 182, 187-189  
 Operadores de conversión, 574-578  
     implícitos vs. explícitos, 574, 578  
     restricciones, 577-578  
 Operadores lógicos, 64-69, 566  
 Operadores relationales, 28, 64-65, 259-261, 566  
 operator, palabra clave, 246  
 OR, operador (!)  
     bitwise, 182, 183, 184-185  
     lógico, 64, 65, 69-69  
 OR, operador, circuito corto (!!), 64, 66-67, 68-69  
 order by, cláusula, 512, 514-515  
 OrderBy( ), 534  
 OrderByDescending( ), 534  
 out, modificador de parámetro, 208-209, 211-212, 222-223, 247, 269, 274  
 OutOfMemoryException, 378  
 OverflowException, 378, 392  
     y checked y unchecked, 386-388  
 override, palabra clave, 315, 318  
 Overriding, método, 315-322  
 Overriding, método y envío en métodos dinámicos, 316
- 
- P**
- 
- Palabras clave, C#  
     contextual, 35, 37  
     reservadas, 35, 37  
 Parámetro(s), 128, 134-137  
     modificadores, 208-216  
     tipo. *Ver* Parámetros de tipo  
     y constructores sobrecargados, 226  
     y métodos sobrecargados, 218, 220-222  
 Parámetros de tipo, 466  
     comparar dos, 497-498  
     relación entre dos, establecer una, 473, 476-477  
     valor por defecto de, crear un, 485-487  
     y limitaciones, 473  
 parámetro de valor, 263, 270  
 params, modificador de parámetros, 213-216  
 Parse( ), métodos, 420, 421-424  
 partial, modificador, 589-591  
 PathTooLongException, 401  
 Peej( ), 395, 583, 584  
 Pila, definición de, 10, 170  
 Polimorfismo, 9, 10  
     e interfaces, 335  
     y métodos sobrecargados, 218, 223  
     y sobrecargar un métodos, tiempo de ejecución, 316, 318  
 Portabilidad, 4, 5, 7, 8, 41  
 Posición, propiedad, 393, 395  
 Pow( ), 77  
 Predicado (LINQ query), 509  
 Programación  
     estructurada, 3, 9  
     lenguajes mezclados, 5, 7, 8, 41  
     orientada a objetos. *Ver* Programación orientada a objetos (POO)  
 Programación orientada a objetos (POO), 3, 9-11, 198  
 Propiedades, 270-274  
     autoimplementadas, 270, 273-274  
     interfaz, 347-349  
     restricciones, 274  
     sólo-lectura o sólo-escritura, crear, 271-272, 275  
     virtual, 318  
     y genéricos, 472, 497  
     y herencia, 292-294, 296  
 Propiedades virtuales e indexadores, 318  
 protected, especificador de acceso, 199, 292, 294-296

public, especificador de acceso, 121, 199-205, 295  
Punteros, 567-571  
Punto flotante  
    literales, 51, 52  
    tipos, 24, 26, 44

## Q

---

Query  
    continuidad, 521-523  
    definición de, 507  
    determinar qué se obtiene con, 509-510, 516-519  
    formato general de, 512-513  
    fuente de datos, 509, 511-512  
    grupo, resultado de, 519-521  
    ordenar resultado de, 515-515  
    palabras clave contextuales, lista de, 512  
    resultados temporales en un, generar, 521-523  
    tipos de datos, relación entre, 511-512  
    unir dos fuentes de datos en un, 525-528  
    valores regresados por, filtro, 509, 513-514  
    variable, 509, 511, 512  
    variable de rango, 509, 511-512, 516  
    variable en un, 523-525  
    y unión de grupos, 531-534  
Query, crear un, 542  
    usar la sintaxis de, 508-510  
    usar los métodos, 536-538  
Query, ejecución, 508, 510-511, 512  
    diferido vs. inmediato, 541-542  
Query, métodos, 534-542  
    correspondientes a palabras clave, 534-535  
    y expresiones lambda para crear queries, usar, 536-538  
Queue<T>, colección, 580, 583-586  
Quicksort, algoritmo, 159, 160, 235, 239-242

## R

---

Read( ), 82-83, 394, 395, 397, 399, 402, 403, 415  
ReadBlock( ), 395  
ReadByte( ), 394, 402-404, 416  
ReadKey( ), 399-400  
ReadLine(), 395, 397, 398  
ReadTimeOut, propiedad, 395  
ReadToEnd( ), 395  
Recolector de basura, 147-148, 218, 588  
    y arreglos, 154  
    y fixed, 570-571  
Recursión, 234-235  
ref, modificador de parámetro, 208-210, 212-213, 222-223,  
    247, 269, 274  
ReferenceEquals( ), 327

Región de memoria en cúmulo, 587  
Reiterador, 591  
Remove( ), 579  
RemoveAt( ), 580  
Replace( ), 435  
Reservar memoria  
    usando new, 126, 146-147, 155  
    usar stackalloc, 587  
Restricciones, 473-485  
    clase base, 473, 474-476, 478, 484  
    constructor, 473, 479-481  
    interfaz, 473, 477-479  
    múltiples, uso, 484-485  
    tipo de referencia, 473, 481-484  
    tipo desnudo, 473, 476-477  
    tipo valor, 473, 481, 483-484  
    y métodos genéricos, 491  
return, declaración, 82, 130-131, 209, 441, 547  
Ritchie, Dennis, 3

## S

---

SByte, estructura, 421  
sbyte, tipo simple, 41, 42, 43, 52  
sealed, palabra clave, 326-327  
SecurityException, 401  
Seek( ), 393, 394, 418-420, 430  
SeekOring, enumeración, 418  
Seguridad, 4, 5, 6, 7, 8  
Selección, declaración, 82  
Select( ), 534, 536-537  
select, cláusula, 509-510, 512, 513, 516-519  
    usar into con, 521-522  
SerializationInfo, 378  
Sensibilidad a las mayúsculas y C#, 20  
set, accesador  
    para un indexador, 263-266, 267, 349  
    para una propiedad, 270, 272, 348  
    para una propiedad autoimplementada, 274  
    usar modificadores de acceso con, 275-277  
SetError( ), 411  
SetIn( ), 411  
SetOut( ), 411  
Sheridan, Mike, 4  
short, tipo simple, 41, 42, 43, 52  
Sign, bandera, 42  
sizeof, operador, 586  
Sobrecarga  
    constructores, 224-231  
    indexadores, 269  
    métodos, 218-224  
    operadores. Ver Operador sobrecargado

**T**

- 
- SQL (Lenguaje Estructurado Query)
- LINQ a SQL, característica de Visual C#, 538
  - y LINQ, 507
  - Sqrt( ), 44, 131, 236
  - Stack<T>, colección, 580
  - stackalloc, 587-588
  - StackOverflowException, 378
  - StackTrace, propiedad de excepción, 376-377
  - static, 19, 236-239, 315, 335
  - Stream, clase, 393, 400, 406, 430
    - métodos, tabla de, 394
    - propiedades, tabla de, 395
  - StreamingContext, 378
  - StreamReader, clase, 396, 406, 409-410, 589
    - y codificación de caracteres, 410
  - StreamWriter, clase, 396, 391-408, 411
    - y codificación de caracteres, 410
  - string, clase, 23, 177-182
    - métodos, 178-180
  - StringBuilder, clase, 182
  - StringReader, clase, 396, 430
  - Stroustrup, Bjarne, 3, 4
  - struc, palabra clave, 356, 357, 473, 481
  - Subclase, 294
  - Substring( ), 181-182, 521
  - Sum( ), método de extensión, 539-540
  - Sun Microsystems, 4
  - Superclase, 294
  - switch, declaración, 82, 87-92
    - y goto, 111
      - y la regla de no interferencia, 89-91, 92
  - System, nomenclatura, 19, 21, 378, 393, 421, 450
  - System.Array.Sort( ), 240
  - System.Collections, 550, 579
  - System.Collections.Generic, 508, 550, 579
  - System.Diagnostics, nomenclatura, 572
  - System.Exception, clase. *Ver Excepción, clase*
  - System.IDisposable, interfaz, 588
  - System.IO, nomenclatura, 393, 394, 400, 450
  - System.IO.Stream, clase. *Ver Stream, clase*
  - System.Linq, nomenclatura, 509, 520, 544, 550
  - System.Linq.Enumerable, clase, 534
  - System.Linq.Expressions.Expression<T>, clase, 538
  - System.Linq.Queryable, clase, 534
  - System.Math, clase, 44, 223
  - System.Nullable<T>, 563
  - System.Object, clase, 327
  - System.Runtime.Serialization, nomenclatura, 378
  - System.Text, nomenclatura, 182
  - System.Type, clase, 562, 563
  - SystemException, 363, 378
  - SystemInvalidOperationException, 564, 565
- 
- TargetSite, propiedad de Exception, 376-377
- TextReader, clase, 394-395, 396, 397, 398, 406, 411, 589
  - métodos de entrada definidos por, tabla, 395
- TextWriter, clase, 394, 396, 398, 406, 411
- this, 148-150, 226-228, 237
  - y métodos de extensión, 542, 543
- throw, 363, 372-374
- Tipo
- anulables. *Ver Anulables, tipos*
  - cerrados vs. construcción abierta, 467
  - extensibilidad, 246
  - identificación, tiempo de ejecución, 561-563
  - inferencia, 490
  - parametrizados, 465
  - parciales, 589-590
  - reglas de promoción, 73-76
  - seguridad y genéricos, 464, 465, 468-471, 476, 490, 494
  - transformar. *Ver Cast.*
  - verificar, 40, 70, 313, 529
- tipo simple decimal, 41, 44-45
- tipo simple decimal literal, 52
- Tipos
- anónimos. *Ver Tipos anónimos.*
  - datos. *Ver Tipo(s) de datos*
  - restringidos, 473. *Ver también Restricciones.*
- Tipo(s) de dato, 23, 26
  - como clase, 122
  - importancia de, 40
  - primitivos, 41
  - referencia, 40-41, 126, 206
  - simple. *Ver Tipos simples.*
  - transformación. *Ver Cast.*
  - valor. *Ver Tipos de valor.*
- Tipos anónimos, 528-535
  - e inicializadores de proyección, 530
  - y join, 528, 529-530, 538
  - y reglas de verificación de, 529
  - y variables de tipo implícito, 530
- Tipos de referencia, 40-41, 126, 147
  - valor por defecto, 143, 485
- Tipos de valor
- encajonar y desencajonar, 329-330
  - .NET, estructuras numéricas, 420-421, 424
  - valor por defecto de, 143, 485
  - y new, 147
  - y ref, 208-210
- Tipos primitivos, 41
- Tipos simples, 41-47
  - tabla de, 41
- ToArray( ), 584
- ToCharArray( ), 523

ToString( ), 327, 328-329, 330, 376-377  
TrimExcess( ), 584  
true, 47, 63, 261  
True y false en C#, 47  
try, bloques, 363-366  
    anidados, 370-372  
typeof, operador, 562-563

## U

---

uint, tipo simple, 41, 42, 43, 52  
    literal, 52  
UInt16, estructura, 421  
UInt32, estructura, 421  
UInt64, estructura, 421  
ulong, tipo simple, 41, 42, 52  
    literal, 52  
unchecked, palabra clave, 386-388  
Unicode, 46, 183, 363, 406  
UnmanagedMemoryStream, clase, 394  
unsafe, palabra clave, uso, 570  
ushort, tipo simple, 41, 42, 43, 52  
using  
    declaración, 402, 588-589  
    directiva, 19, 21, 452-455

## V

---

Value, propiedad, 564  
var, palabra clave, 24, 57, 169, 170, 174, 486, 510, 512  
Variable(s)  
    alcance y tiempo de vida, 58-61  
    capturar, 443  
    const, 589  
    declaración, 22-23, 24, 29-30, 55, 58, 60  
    definición de, 21  
    estáticas, 236-238  
    externa, 443  
    Inicialización dinámica, 56  
    Inicializar, 56, 60  
    instancia. *Ver* variables de instancia  
    local, 55  
    miembro, 10  
    nombres, reglas aplicables, 35-36  
    query, 509, 511, 512  
    rango, 509, 511-12, 516  
    referencia. *Ver* Variables de referencia  
    tipo implícito, 6, 24, 57-58, 169, 530  
Variables de instancia  
    acceder, 123, 130  
    base para accesar, ocultas, usar, 303-304  
    como únicas de su objeto, 123, 124-125

declarar, 122  
definición de, 10, 121  
ocultar, 150  
this, para accesar ocultas, usar, 150  
Variables de referencia  
    asignar referencias de variables derivadas a clase base, 313-315  
    como argumentos para un método, 207-208  
    declarar, 126  
    modificar con ref y out, 212-213  
    y asignación, 126-127  
virtual, palabra clave, 315, 318, 323  
Visual C#, 12  
    LINQ a SQL, característica, 538  
Visual C++ 2008 Express, 17  
Visual Studio IDE, 12-16, 17  
void, 19, 128  
    métodos, 130-131  
volatile, modificador, 589  
vsvars.bat, archivo, 17

## W

---

Warth, Chris, 4  
Where( ), 534-535, 536-537  
where, cláusula  
    en un query LINQ, 509, 512, 513-514  
    y parámetros de tipo, 473, 474, 477, 478, 481, 484, 485  
while, loop, 82, 100-101  
Write( ), 24, 394, 296, 398, 404, 415  
WriteByte( ), 394, 404  
WriteLine( ), 20, 23, 24, 26, 396, 398, 420  
    versión de salida con formato, 48-49  
    WriteTimeout, propiedad, 395  
    y ToString(), 328, 376-377  
    y valores bool, 47  
    y valores char, 46

## X

---

XML  
    archivos, usar LINQ con, 507  
    comentario, 20  
XOR, operador OR exclusivo (^)

## Y

---

yield, palabra clave contextual, 591  
bitwise, 182, 183, 185-186  
lógico, 64, 65

# Habilidades esenciales – ¡Fácilmente!

Permite que el maestro de programación y autor de *bestsellers* Herb Schildt te enseñe los fundamentos de C#, el lenguaje de programación premier de Microsoft para .NET Framework. Comenzarás por aprender a crear, compilar y ejecutar un programa en C#. Después aprenderás a usar los diferentes tipos de datos, operadores, declaraciones de control, métodos, clases y objetos. También aprenderás sobre herencia, interfaces, propiedades, indexadores, excepciones, eventos, nomenclaturas, genéricos y mucho más. Por supuesto, incluye las nuevas características de C# 3.0 como LINQ, expresiones *lambda* y tipos anónimos. Comienza a programar hoy en C# con la ayuda de este libro práctico y de rápida lectura.

## Diseñado para aprender fácilmente

- E Habilidades clave y conceptos: Lista de las habilidades específicas que obtendrás en cada capítulo.
- E Pregunta al experto: Sección de preguntas y respuestas que ofrecen información adicional y consejos de utilidad.
- E Prueba esto: Ejercicios prácticos para aplicar las habilidades adquiridas.
- E Notas: Información adicional relativa al tema en estudio.
- E Autoexamen: Ensayos para medir tu conocimiento al finalizar cada capítulo.
- E Sintaxis: Ejemplo de código y comentarios que describen las técnicas de programación que se muestran.

Código listo para utilizarse en [www.mcgraw-hill-educacion.com](http://www.mcgraw-hill-educacion.com)  
(busca por ISBN)

**Herbert Schildt** es autoridad reconocida en C#, C, C++ y Java. Sus libros de programación han vendido más de 3.5 millones de ejemplares en todo el mundo y han sido traducidos a los idiomas más importantes del planeta. Herb es autor de numerosos *bestsellers*, incluyendo *Java Manual de referencia, séptima edición*, *Java Soluciones de programación* y *C++ Soluciones de programación*. Conoce más en [HerbSchildt.com](http://HerbSchildt.com)

The McGraw-Hill Companies



Educación

Visite nuestra página WEB  
[www.mcgraw-hill-educacion.com](http://www.mcgraw-hill-educacion.com)