

DOUGLAS BELL & MIKE PARR

C#

PARA

*Estudiantes*

PEARSON



# C# PARA *Estudiantes*



**DOUGLAS BELL  
MIKE PARR**

**C#**  
**PARA**  
*Estudiantes*

TRADUCCIÓN

**Alfonso Vidal Romero Elizondo**

*Ingeniero en Computación*

*Instituto Tecnológico y de Estudios Superiores  
de Monterrey, campus Monterrey*

REVISIÓN TÉCNICA

**Jakeline Marcos Abed**

**Yolanda Martínez Treviño**

*Departamento de Ciencias Computacionales*

*División de Mecatrónica y Tecnologías de Información  
Tecnológico de Monterrey, Campus Monterrey*

**Eduardo Ramón Lemus Velázquez**

*Escuela de Ingeniería*

*Universidad Panamericana, México*

**Addison Wesley**

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador  
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

**BELL, DOUGLAS y PARR, MIKE**

**C# para estudiantes. Primera edición**

PEARSON EDUCACIÓN, México, 2010

ISBN: 978-607-32-0328-9

Área: Informática

Formato: 18.5 × 23.5 cm

Páginas: 464

Authorized translation from the English language edition, entitled *C# FOR STUDENTS – REVISED 01 Edition*, by *Douglas Bell & Mike Parr* published by Pearson Education Limited, United Kingdom © 2009. All rights reserved.  
ISBN 9780273728207

Traducción autorizada de la edición en idioma inglés, titulada *C# FOR STUDENTS – REVISED 01 Edition*, por *Douglas Bell & Mike Parr* publicada por Pearson Education Limited, United Kingdom © 2009. Todos los derechos reservados.

Esta edición en español es la única autorizada.

**Edición en español**

Editor: Luis Miguel Cruz Castillo  
e-mail: luis.cruz@pearsoned.com  
Editor de desarrollo: Bernardino Gutiérrez Hernández  
Supervisor de producción: Rodrigo Romero Villalobos

PRIMERA EDICIÓN, 2011

D.R. © 2011 por Pearson Educación de México, S.A. de C.V.  
Atacomulco 500-5o. piso  
Col. Industrial Atoto  
53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. núm. 1031.

Addison Wesley es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotográfico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

**Addison Wesley**  
es una marca de



ISBN VERSIÓN IMPRESA: 978-607-32-0328-9  
ISBN VERSIÓN E-BOOK: 978-607-32-0329-6  
ISBN E-CHAPTER: 978-607-32-0330-2

PRIMERA IMPRESIÓN  
Impreso en México. *Printed in Mexico.*  
1 2 3 4 5 6 7 8 9 0 - 14 13 12 11

# Contenido breve

Contenido	vii
Prefacio	xvii
1. Antecedentes sobre C#	1
2. El entorno de desarrollo de C#	7
3. Introducción a los gráficos	24
4. Variables y cálculos	37
5. Métodos y argumentos	59
6. Uso de los objetos	92
7. Selección	113
8. Repetición	143
9. Depuración	162
10. Creación de clases	173
11. Herencia	196
12. Cálculos	211
13. Estructuras de datos: cuadros de lista y listas	228
14. Arreglos	240
15. Arreglos bidimensionales (matrices)	262
16. Manipulación de cadenas de caracteres	275
17. Excepciones	296
18. Archivos	312
19. Programas de consola	336
20. El diseño orientado a objetos	351
21. Estilo de programación	375
22. La fase de pruebas	388
23. Interfaces	402
24. Polimorfismo	408

<b>Apéndices</b>	417
<b>Bibliografía</b>	433
<b>Índice</b>	434



# Contenido

Prefacio	xvii
<b>1. Antecedentes sobre C#</b>	<b>1</b>
La historia de C#	1
El marco de trabajo .NET de Microsoft	2
¿Qué es un programa?	2
Fundamentos de programación	4
Errores comunes de programación	5
Resumen	5
Ejercicios	5
Soluciones a las prácticas de autoevaluación	6
<b>2. El entorno de desarrollo de C#</b>	<b>7</b>
Introducción	7
Instalación y configuración	7
Cree su primer programa	8
Los controles en tiempo de diseño	13
Los eventos y el control <code>Button</code>	15
Apertura de un proyecto existente	17
Documentación de los valores de las propiedades	17
Errores en los programas	18
Funciones del editor	19
El cuadro de mensajes	20
Ayuda	21
Fundamentos de programación	21
Errores comunes de programación	21
Secretos de codificación	21

Nuevos elementos del lenguaje	22
Nuevas características del IDE	22
Resumen	22
Ejercicios	22
Soluciones a las prácticas de autoevaluación	23
<b>3. Introducción a los gráficos</b>	<b>24</b>
Introducción	24
Objetos, métodos, propiedades, clases: una analogía	24
Nuestro primer dibujo	25
Creación del programa	26
El sistema de coordenadas de gráficos	27
Explicación del programa	28
Métodos para dibujar	30
Colores	31
El concepto de secuencia y las instrucciones	33
Adición de significado mediante el uso de comentarios	33
Fundamentos de programación	34
Errores comunes de programación	35
Secretos de codificación	35
Nuevos elementos del lenguaje	35
Nuevas características del IDE	35
Resumen	35
Ejercicios	35
Soluciones a las prácticas de autoevaluación	36
<b>4. Variables y cálculos</b>	<b>37</b>
Introducción	37
La naturaleza de <code>int</code>	38
La naturaleza de <code>double</code>	38
Declaración de variables	39
La instrucción de asignación	42
Cálculos y operadores	43
Los operadores aritméticos	44
El operador <code>%</code>	46
Unión de cadenas con el operador <code>+</code>	47
Conversiones entre cadenas y números	49
Cuadros de texto y etiquetas	50
Conversiones entre números	52
Función de las expresiones	54
Fundamentos de programación	54
Errores comunes de programación	54
Secretos de codificación	55
Nuevos elementos del lenguaje	55

Nuevas características del IDE	55
Resumen	55
Ejercicios	56
Soluciones a las prácticas de autoevaluación	58
<b>5. Métodos y argumentos</b>	<b>59</b>
Introducción	59
Creación de métodos propios	60
Nuestro primer método	60
Cómo invocar métodos	62
Cómo pasar argumentos	63
Parámetros y argumentos	64
Un método para dibujar triángulos	65
Variables locales	68
Conflictos de nombres	69
Métodos para manejar eventos	70
La instrucción <code>return</code> y los resultados	71
Construcción de métodos a partir de otros métodos	73
Transferencia de argumentos por referencia	76
Los argumentos <code>out</code> y <code>ref</code>	77
Un ejemplo con <code>out</code>	77
Un ejemplo con <code>ref</code>	80
Un método de intercambio con <code>ref</code>	81
La palabra clave <code>this</code> y los objetos	82
Sobrecarga de métodos	83
Transferencia de objetos a los métodos	85
Fundamentos de programación	85
Errores comunes de programación	85
Secretos de codificación	86
Nuevos elementos del lenguaje	87
Nuevas características del IDE	87
Resumen	87
Ejercicios	87
Soluciones a las prácticas de autoevaluación	90
<b>6. Uso de los objetos</b>	<b>92</b>
Introducción	92
Variables de instancia	93
El constructor del formulario	96
La clase <code>TrackBar</code>	98
La palabra clave <code>using</code> y los espacios de nombres	101
Miembros, métodos y propiedades	102
La clase <code>Random</code>	102
La clase <code>Timer</code>	106

Fundamentos de programación	109
Errores comunes de programación	109
Secretos de codificación	109
Nuevos elementos del lenguaje	109
Nuevas características del IDE	110
Resumen	110
Ejercicios	110
Soluciones a las prácticas de autoevaluación	112
<b>7. Selección</b>	<b>113</b>
Introducción	113
La instrucción <code>if</code>	114
<code>if... else</code>	116
Operadores de comparación	118
And, or, not	123
Instrucciones <code>if</code> anidadas	126
La instrucción <code>switch</code>	127
Variables booleanas	131
Fundamentos de programación	135
Errores comunes de programación	135
Secretos de codificación	136
Nuevos elementos del lenguaje	136
Resumen	137
Ejercicios	137
Soluciones a las prácticas de autoevaluación	140
<b>8. Repetición</b>	<b>143</b>
Introducción	143
<code>while</code>	144
<code>for</code>	148
And, or, not	150
<code>do... while</code>	152
Ciclos anidados	153
Combinación de las estructuras de control	155
Fundamentos de programación	155
Errores comunes de programación	155
Secretos de codificación	156
Nuevos elementos del lenguaje	156
Resumen	157
Ejercicios	157
Soluciones a las prácticas de autoevaluación	159
<b>9. Depuración</b>	<b>162</b>
Introducción	162
Uso del depurador	164

Ejemplo práctico de depuración	167
Errores comunes	168
Errores comunes de programación	172
Nuevas características del IDE	172
Resumen	172
Ejercicio	172
<b>10. Creación de clases</b>	<b>173</b>
Introducción	173
Cómo diseñar una clase	174
Variables <code>private</code>	176
Métodos <code>public</code>	177
Propiedades	179
¿Método o propiedad?	182
Constructores	183
Múltiples constructores	183
Métodos <code>private</code>	185
Operaciones sobre objetos	185
Destrucción de objetos	187
Métodos y propiedades <code>static</code>	187
Fundamentos de programación	188
Errores comunes de programación	190
Secretos de codificación	191
Nuevos elementos del lenguaje	192
Resumen	192
Ejercicios	192
Soluciones a las prácticas de autoevaluación	194
<b>11. Herencia</b>	<b>196</b>
Introducción	196
Uso de la herencia	197
<code>protected</code>	198
Elementos adicionales	199
Redefinición	200
Diagramas de clases	201
La herencia en acción	202
<code>base</code>	202
Constructores	203
Clases abstractas	205
Fundamentos de programación	206
Errores comunes de programación	208
Nuevos elementos del lenguaje	208
Resumen	208

Ejercicios	209
Soluciones a las prácticas de autoevaluación	210
<b>12. Cálculos</b>	<b>211</b>
Introducción	211
Aplicación de formato a los números	212
Funciones y constantes de la biblioteca matemática	214
Constantes	215
Ejemplo práctico: dinero	216
Ejemplo práctico: iteración	218
Gráficas	219
Excepciones	222
Fundamentos de programación	223
Errores comunes de programación	223
Resumen	223
Ejercicios	224
Soluciones a las prácticas de autoevaluación	227
<b>13. Estructuras de datos: cuadros de lista y listas</b>	<b>228</b>
Introducción	228
Listas	229
Adición de elementos a una lista	229
La longitud de una lista	230
Índices	230
Eliminación de elementos de una lista	233
Inserción de elementos en una lista	233
Búsquedas rápidas	233
Operaciones aritméticas en cuadros de lista	234
Búsquedas detalladas	236
Listas genéricas	237
Fundamentos de programación	238
Errores comunes de programación	238
Nuevos elementos del lenguaje	238
Resumen	239
Ejercicios	239
Soluciones a las prácticas de autoevaluación	239
<b>14. Arreglos</b>	<b>240</b>
Introducción	240
Creación de un arreglo	242
Índices	242
Longitud de los arreglos	245
Paso de arreglos como parámetros	245
Uso de constantes	246
Inicialización de arreglos	247

Un programa de ejemplo	248
Búsqueda rápida (accesos directos)	250
Búsqueda detallada	251
Arreglos de objetos	252
Fundamentos de programación	254
Errores comunes de programación	255
Secretos de codificación	255
Resumen	256
Ejercicios	256
Soluciones a las prácticas de autoevaluación	260
<b>15. Arreglos bidimensionales (matrices)</b>	262
Introducción	262
Declaración de matrices	263
Índices	263
Longitud de la matriz	265
Paso de matrices como parámetros	265
Constantes	266
Inicialización de matrices	267
Un programa de ejemplo	267
Fundamentos de programación	269
Errores comunes de programación	270
Resumen	270
Ejercicios	271
Soluciones a las prácticas de autoevaluación	274
<b>16. Manipulación de cadenas de caracteres</b>	275
Introducción	275
Uso de cadenas de caracteres: un recordatorio	275
Indexación de cadenas	277
Caracteres dentro de las cadenas de caracteres	277
Una observación sobre el tipo <code>char</code>	278
Métodos y propiedades de la clase <code>String</code>	278
Comparación de cadenas	279
Corrección de cadenas	280
Análisis de cadenas	282
Expresiones regulares	286
Un ejemplo de procesamiento de cadenas	288
Ejemplo práctico: Frasier	289
Fundamentos de programación	292
Errores comunes de programación	292
Secretos de codificación	292
Nuevos elementos del lenguaje	292
Nuevas características del IDE	292

Resumen	293
Ejercicios	293
Soluciones a las prácticas de autoevaluación	295
<b>17. Excepciones</b>	<b>296</b>
Introducción	296
La jerga de las excepciones	298
Un ejemplo con <code>try-catch</code>	298
Uso del objeto de excepción	301
Clasificación de las excepciones	302
Bloques <code>catch</code> múltiples	303
Búsqueda de un bloque <code>catch</code>	304
Lanzar una excepción: una introducción	306
Posibilidades del manejo de excepciones	307
<code>finally</code>	307
Fundamentos de programación	308
Errores comunes de programación	309
Secretos de codificación	309
Nuevos elementos del lenguaje	309
Nuevas características del IDE	309
Resumen	310
Ejercicios	310
Soluciones a las prácticas de autoevaluación	311
<b>18. Archivos</b>	<b>312</b>
Introducción	312
Fundamentos de los flujos	313
Las clases <code>StreamReader</code> y <code>StreamWriter</code>	313
Operaciones de salida con archivos	314
Operaciones de entrada con archivos	316
Operaciones de búsqueda en archivos	318
Archivos y excepciones	321
Mensajes y cuadros de diálogo	323
Cuadros de diálogo para manejo de archivos	325
Creación de menús	326
La clase <code>Directory</code>	330
Fundamentos de programación	332
Errores comunes de programación	332
Secretos de codificación	332
Nuevos elementos del lenguaje	332
Nuevas características del IDE	333
Resumen	333
Ejercicios	333
Soluciones a las prácticas de autoevaluación	334



<b>19. Programas de consola</b>	336
Introducción	336
Nuestro primer programa de consola	337
El símbolo del sistema: <code>cd</code> y <code>dir</code>	339
Formas de ejecutar programas	340
Uso de clases en aplicaciones de consola	342
Argumentos de la línea de comandos	342
Secuencias de comandos y redirección de la salida	344
Secuencias de comandos y archivos de procesamiento por lotes	346
Fundamentos de programación	347
Errores comunes de programación	347
Secretos de codificación	347
Nuevos elementos del lenguaje	347
Nuevas características del IDE	347
Resumen	347
Ejercicios	348
Soluciones a las prácticas de autoevaluación	349
 <b>20. El diseño orientado a objetos</b>	 351
Introducción	351
El problema del diseño	352
Identificación de los objetos, métodos y propiedades	352
Ejemplo práctico de diseño	358
En búsqueda de la reutilización	365
¿Composición o herencia?	365
Lineamientos para el diseño de clases	370
Resumen	371
Ejercicios	372
Soluciones a las prácticas de autoevaluación	373
 <b>21. Estilo de programación</b>	 375
Introducción	375
Estructura del programa	376
Comentarios	377
Uso de constantes	378
Clases	379
Instrucciones <code>if</code> anidadas	380
Ciclos anidados	382
Condiciones complejas	384
Documentación	386
Errores comunes de programación	386
Resumen	386
Ejercicios	387

<b>22. La fase de pruebas</b>	<b>388</b>
Introducción	388
Especificaciones de los programas	389
Prueba exhaustiva	390
Prueba de la caja negra (funcional)	390
Prueba de la caja blanca (estructural)	393
Inspecciones y recorridos	395
Recorrer paso a paso las instrucciones del programa	396
Verificación formal	396
Desarrollo incremental	397
Fundamentos de programación	397
Resumen	398
Ejercicios	398
Soluciones a las prácticas de autoevaluación	400
 <b>23. Interfaces</b>	 <b>402</b>
Introducción	402
Interfaces para el diseño	402
Interfaces e interoperabilidad	405
Fundamentos de programación	406
Errores comunes de programación	406
Nuevos elementos del lenguaje	406
Resumen	406
Ejercicios	407
 <b>24. Polimorfismo</b>	 <b>408</b>
Introducción	408
El polimorfismo en acción	409
Fundamentos de programación	414
Errores comunes de programación	415
Nuevos elementos del lenguaje	415
Resumen	415
Ejercicios	415
 <b>Apéndices</b>	 <b>417</b>
A Componentes de la biblioteca seleccionados	417
B Palabras reservadas (clave)	432
 Bibliografía	 433
Índice	434

# Prefacio

## ● Este libro está dirigido a principiantes

---

Si nunca ha programado —si es un completo principiante—, este libro es para usted. No necesita tener conocimientos previos sobre programación, ya que en el texto se explican los conceptos desde cero con un estilo simple y directo para conseguir la máxima claridad. El libro está dirigido a los estudiantes universitarios de primer nivel, pero también es apropiado para los principiantes autodidactas.

## ● ¿Por qué C#?

---

Sin duda C# es uno de los mejores lenguajes de programación que podemos aprender y usar en el siglo XXI, debido a que:

- C# continúa con la tradición de la familia de lenguajes que incluye a C, C++ y Java.
- Los lenguajes orientados a objetos representan la metodología más reciente y exitosa en materia de programación. C# es completamente orientado a objetos.
- C# es un lenguaje de propósito general: todo lo que Visual Basic, C++ y Java pueden hacer, es posible en C#.
- C# obtiene la mayor parte de su funcionalidad de una biblioteca de componentes proporcionados por el marco de trabajo (Framework) de .NET.

## ● Requerimientos

---

Para aprender a programar en C# necesitará una PC en la que pueda ejecutar el software que le permita preparar y ejecutar programas de C# de manera conveniente, esto es, un entorno de desarrollo. Microsoft proporciona dos versiones del software:

1. Visual C# 2008 Express Edition (para C# solamente).
2. Visual Studio 2008 (con soporte para C# y otros lenguajes).

La primera opción está disponible como descarga gratuita, y es completamente adecuada para desarrollar los programas que se analizan en este libro.

## ● **Metodología de la obra**

---

En este libro explicaremos desde los primeros capítulos cómo utilizar objetos. Nuestra metodología consiste en empezar por brindar definiciones sobre los conceptos de variables, asignaciones y métodos, para después usar objetos creados a partir de clases de biblioteca. Posteriormente veremos cómo utilizar las estructuras de control de selección y de ciclo. Por último, mostraremos de qué manera escribir clases propias.

Para asegurarnos de que la facilidad de uso fuera el elemento más importante, utilizamos gráficos desde el principio, porque consideramos que son divertidos, interesantes y capaces de demostrar con claridad todos los principios importantes de la programación. Esto no quiere decir que hayamos ignorado los programas que trabajan con texto en sus operaciones de entrada y de salida; también los incluimos.

Los programas que examinaremos en este libro utilizan muchas de las características de las interfaces gráficas de usuario (GUI), como botones y cuadros de texto. Además, explicaremos cómo escribir programas de línea de comandos (programas de consola) en C#.

Presentaremos las nuevas ideas con cuidado, una por una, en vez de introducirlas todas juntas. Por ejemplo, hay un capítulo completo para enseñarle a escribir métodos. Con este fin, abordaremos los conceptos simples en los primeros capítulos, y los más sofisticados en los posteriores.

## ● **Contenido**

---

Este libro explica las ideas fundamentales de la programación:

- Variables.
- Asignación.
- Entrada y salida mediante el uso de una interfaz gráfica de usuario (GUI).
- Cálculos.
- Repeticiones.
- Cómo seleccionar entre alternativas.

Además indica cómo utilizar números y cadenas de caracteres, y describe los arreglos. Todos estos temas son fundamentales, sin importar el tipo de programación que usted lleve a cabo. Asimismo, analiza con detalle los principios de la programación orientada a objetos: cómo utilizar objetos, escribir clases, métodos y propiedades; y cómo aprovechar las clases de biblioteca. De igual manera, aborda algunos de los aspectos más sofisticados de la programación orientada a objetos, incluyendo la herencia, el polimorfismo y las interfaces.

## ● Limitaciones

Este libro se limita a tratar los aspectos esenciales de C# y no explica todas las especificidades relacionadas con este lenguaje. En consecuencia, el lector no tendrá que lidiar con detalles innecesarios y podrá concentrarse en dominar el lenguaje C# y la programación en general.

## ● UML

Actualmente el Lenguaje Unificado de Modelado (UML) es la notación dominante para describir programas computacionales. En este libro utilizamos elementos de UML de manera selectiva y en donde sea apropiado.

## ● Aplicaciones

Las computadoras se utilizan en muchas aplicaciones. En este sentido, el libro utiliza ejemplos de todas las áreas, incluyendo:

- Juegos.
- Procesamiento de información.
- Cálculos científicos.

El lector puede optar por concentrarse en las áreas de aplicación que sean de su interés e ignorar las demás.

## ● Ventajas de los ejercicios

Aunque usted leyera este libro varias veces hasta que pudiera recitarlo al revés, no por ello podría escribir programas. El trabajo práctico relacionado con la escritura de programas es vital para obtener fluidez y confianza en la programación.

Al final de cada capítulo se proponen varios ejercicios; le invitamos a que realice algunos de ellos para mejorar sus habilidades de programador.

También se presentan breves prácticas de autoevaluación a lo largo del libro, para que se cerciore de haber entendido las cosas de manera apropiada. Al final de cada capítulo se proporcionan las soluciones correspondientes.

## ● Diviértase

La programación —especialmente la que se realiza en C#— es creativa e interesante, por lo tanto, ¡diviértase!

## ● Visite nuestro sitio Web

---

El sitio Web de este libro incluye:

- El código de todos los programas incluidos en el texto.
- Recursos adicionales para instructores.

Para acceder al sitio Web visite la siguiente dirección:  
[www.pearsoneducacion.net/bell](http://www.pearsoneducacion.net/bell)

## ● Novedades en esta edición

---

La versión más reciente de C# es la 2008. Este libro utiliza lo mejor de las novedades presentadas en ella, pero no incluimos todas, pues consideramos que algunas no son apropiadas en un libro orientado a principiantes. Microsoft también ofrece un entorno de desarrollo llamado Visual C# 2008 Express, el cual soporta la versión de C# que utilizamos aquí.

Los cambios realizados respecto de la edición anterior de este libro son:

Capítulo 2. El entorno de desarrollo de C#.

Se modificó para explicar cómo utilizar Visual C# 2008 Express.

Capítulo 13. Estructuras de datos: cuadros de lista y listas.

Las listas de arreglos se reemplazaron por listas simples. Además, se amplió la explicación sobre dichos elementos, se utilizaron listas genéricas y se introdujo la instrucción **foreach**.

Capítulo 24. Polimorfismo.

Se utilizó una lista genérica junto con la instrucción **foreach**. Esto elimina la necesidad de realizar conversiones de tipos (*casting*), con lo cual el capítulo es más conciso y útil.

# Antecedentes sobre C#

## En este capítulo conoceremos:

- cómo y por qué surgió C#;
- la tecnología del marco de trabajo .NET (Framework .NET) de Microsoft;
- conceptos preliminares de programación.

## ● La historia de C#

---

Los programas de computación consisten en una serie de instrucciones que obedecen las computadoras. El objetivo de las instrucciones es indicarles cómo realizar una tarea (por ejemplo, ejecutar un juego, enviar un mensaje de correo electrónico, etc.). Las instrucciones están escritas en un estilo particular, acorde con las reglas del lenguaje de programación que elija el programador. Hay cientos de lenguajes de programación, pero sólo unos cuantos han dejado huella, volviéndose muy populares. La historia de los lenguajes de programación es evolutiva, y aquí veremos las raíces de C#. Los nombres de los lenguajes anteriores no son importantes, sin embargo, los mencionaremos para que el lector tenga una idea más amplia sobre el tema.

Durante la década de 1960 se creó un lenguaje de programación llamado Algol 60 (el término “Algol” proviene de la palabra “algoritmo” en alusión a la serie de pasos que pueden llevarse a cabo para resolver un problema). Este lenguaje fue popular en los círculos académicos, pero los principios en que estaba basado persistieron mucho más tiempo que su uso. En esa época eran más populares otros lenguajes, como el COBOL para el procesamiento de datos, y el Fortran para el trabajo científico. En el Reino Unido se creó una versión extendida de Algol 60 (CPL, o lenguaje de programación combinado), cuyo nombre se simplificó casi de inmediato a CPL básico, o BCPL.

Poco tiempo después Dennis Ritchie y sus colaboradores, investigadores de los laboratorios estadounidenses Bell, transformaron el BCPL en un lenguaje llamado B, que posteriormente fue mejorado hasta convertirse en C durante los años setenta. C resultó enormemente popular: se utilizó para escribir el sistema operativo UNIX y, mucho después, Linux Torvalds lo empleó también para crear una versión de UNIX —conocida como LINUX— para PC.

El siguiente momento importante fue cuando Stroustrup, otro investigador de los Laboratorios Bell, creó C++ durante la década de 1980. Este lenguaje permitió la creación y reutilización de secciones independientes de código, en un estilo conocido como “programación orientada a objetos” (en C podía usarse el operador ++ para sumar uno a un elemento, de ahí que C++ sea uno más que C).

C++ sigue siendo popular, pero es difícil de usar pues requiere mucho estudio. Fue hacia 1995 cuando Sun Microsystems produjo Java, un lenguaje fuertemente basado en objetos, pero más simple que C++. También tenía la ventaja de poder ejecutarse en muchos tipos de computadoras (PC con Microsoft Windows, la Apple Mac, etc.). Java es bastante utilizado todavía.

En 2002 Microsoft anunció la aparición del lenguaje C#, similar a C++ y Java, pero mejorado (en música el símbolo # significa “un semitono más alto”). Este desarrollo era parte importante de la iniciativa “punto net” de Microsoft, que describiremos a continuación. Debido a la similitud entre C# y sus predecesores, al aprenderlo los lenguajes C, C++ y Java le serán más comprensibles si alguna vez necesita utilizarlos.

Por supuesto, esta historia sintetizada de C# se enfoca sólo en la rama evolutiva que corresponde a C, C++ y Java. En 1964 comenzó otra notable ramificación con la invención de un lenguaje de programación para principiantes denominado BASIC, el cual evolucionó hasta convertirse en lo que se conoce hoy como Visual Basic, y que también forma parte de “punto net”.

### ● El marco de trabajo .NET de Microsoft

---

En 2002 Microsoft introdujo un importante producto, conocido como Marco de trabajo .NET (o *Framework .NET*) Las principales características de este marco de trabajo son:

- Incluye los lenguajes de programación C#, Visual Basic y C++.
- Cuenta con herramientas que ayudan a los programadores a crear sitios Web interactivos, como los que se utilizan para el comercio electrónico. Microsoft considera que Internet es crucial, de aquí que haya denominado esta tecnología como .NET (en otras palabras, “punto red”).
- Existe la posibilidad de que .NET esté disponible para otros sistemas operativos y no sólo Microsoft Windows.
- Permite la creación de software a partir de componentes (“objetos”) que pueden difundirse a través de una red.

### ● ¿Qué es un programa?

---

En esta sección trataremos de dar al lector una idea acerca de qué es un programa. Una forma de comprenderlo es mediante el uso de analogías con recetas de cocina, partituras musicales y patrones de bordado. Incluso las instrucciones en una botella de champú para el cabello constituyen un programa simple:

```
mojar el cabello
aplicar el champú
dar masaje para incorporar el champú
enjuagar
```



Este programa es una lista de instrucciones destinadas a un ser humano, pero demuestra un aspecto importante de los programas para computadora: son series de instrucciones que se llevan a cabo de manera secuencial, empezando por la primera y avanzando ordenadamente por las subsecuentes hasta completar la secuencia de instrucciones. Las recetas de cocina, las partituras musicales y los patrones de bordado son similares, en tanto constituyen listas de instrucciones que se llevan a cabo de manera secuencial. En el caso de un patrón de bordado, por ejemplo, la máquina encargada de llevarlo a cabo recibe un programa de instrucciones y lo pone en práctica (o lo “ejecuta”). Eso precisamente es una computadora: una máquina que lleva a cabo de manera automática una secuencia de instrucciones o programa (de hecho, si el programador comete un error al determinar las instrucciones, es probable que la computadora realice una tarea incorrecta). El conjunto de instrucciones disponibles para que una computadora las lleve a cabo suele componerse de:

- ingresar un número (o alimentación, *input*);
- ingresar algunos caracteres (letras y dígitos);
- dar como resultado algunos caracteres (*output*);
- realizar un cálculo;
- mostrar un número como resultado;
- mostrar una imagen gráfica en la pantalla;
- responder al clic de un botón que se encuentra en la pantalla.

La labor de programación consiste en seleccionar de esta lista aquellas instrucciones que lleven a cabo la tarea requerida. Estas instrucciones se escriben en un lenguaje especializado, conocido como lenguaje de programación, y C# es uno de ellos. Aprender a programar significa conocer las herramientas del lenguaje de programación y saber cómo combinarlas para realizar aquello que se desea. El ejemplo de las partituras musicales ilustra otro aspecto de los programas: en las piezas musicales es común que se repitan algunas secciones, digamos el coro. Gracias a una forma de notación específica, las partituras evitan que el compositor tenga que duplicar las partes de la partitura que se repiten. Lo mismo se aplica en los programas: a menudo se da el caso de tener que repetir cierta acción; en un programa de procesamiento de texto, por ejemplo, buscar las apariciones de una palabra dentro de un párrafo. La repetición (o iteración) es común en los programas, y C# cuenta con instrucciones especiales para llevar a cabo estas tareas.

Por su parte, ciertas recetas de cocina suelen indicar algo así como: “si no tiene chícharos frescos utilice congelados”. Esto ilustra otro aspecto de los programas: a menudo llevan a cabo una evaluación antes de realizar una de dos tareas dependiendo del resultado. A esto se le conoce como selección y, al igual que con la repetición, C# cuenta con elementos especiales para realizar esta tarea.

Si alguna vez ha utilizado una receta para preparar un platillo, es muy probable que haya llegado hasta cierto paso sólo para descubrir que tiene que consultar otra receta. Por ejemplo, tal vez tenga que averiguar cómo se cocina el arroz para combinarlo con el resto del platillo; la preparación del arroz se ha separado como una subtarea. Esta forma de escribir instrucciones tiene una importante analogía en la programación, conocida como métodos en C# y en otros lenguajes orientados a objetos. Los métodos se utilizan en todos los lenguajes de programación, aunque algunas veces tienen otros nombres, como funciones, procedimientos, subrutinas o subprogramas.

Los métodos son subtarefas, y se les llama así debido a que constituyen procedimientos para realizar algo. El uso de métodos contribuye a simplificar actividades complejas.

Pensemos ahora en cómo se prepara el mexicanísimo mole. Hasta hace algunos años, la receta nos exigía comprar chiles, frutas y especias secos, para luego freírlos y molerlos. Sin embargo, en la actualidad podemos comprar la pasta ya preparada. Nuestra tarea se ha simplificado. En este caso la analogía con la programación estriba en que la tarea se facilita si existe la posibilidad de seleccionar entre un conjunto de “objetos” prefabricados, como botones, barras de desplazamiento y bases de datos. C# incluye un gran conjunto de objetos que podemos incorporar en nuestros programas en vez de tener que crear todo desde cero.

Para resumir, los programas son listas de instrucciones que una computadora puede obedecer de manera automática. Los programas consisten en combinaciones de:

- secuencias;
- repeticiones;
- selecciones;
- métodos;
- objetos preelaborados y listos para su uso;
- objetos que el mismo programador escribe.

Todos los lenguajes de programación modernos comparten estas características.

#### **PRÁCTICAS DE AUTOEVALUACIÓN**

---

**1.1** He aquí algunas instrucciones para calcular el sueldo de un empleado:

```
obtener el número de horas trabajadas
calcular el sueldo
imprimir recibo de pago
restar deducciones por enfermedad
```

¿Existe algún error importante en la secuencia anterior? ¿Cuál?

**1.2** Modifique la expresión:

```
dar masaje para incorporar el champú
```

para expresarla de manera más detallada, incorporando el concepto de repetición.

**1.3** El siguiente es un aviso antes de subirse a la montaña rusa de un parque de diversiones:

```
¡Sólo pueden subir a este juego personas mayores de 8 años o menores de 70
años!
```

¿Hay algún problema con el aviso? ¿Cómo podría escribirlo para mejorarlo?

### **Fundamentos de programación**

- Los programas consisten de instrucciones combinadas con subtarefas y los conceptos de secuencia, selección y repetición.
- La tarea de programación se simplifica si podemos utilizar componentes prefabricados.

## Errores comunes de programación

Muchas veces pueden filtrarse errores humanos en los programas, por ejemplo, colocar las instrucciones en un orden incorrecto.

## Resumen

- C# es un lenguaje orientado a objetos, derivado de C++ y Java.
- El marco de trabajo (o *framework*) .NET de Microsoft es un producto importante, que incluye los lenguajes C#, C++ y Visual Basic.
- Los programas son listas de instrucciones que las computadoras obedecen de manera automática.
- En la actualidad la principal tendencia en la práctica de la programación es la metodología orientada a objetos (POO), y C# la soporta en su totalidad.

## EJERCICIOS

- 1.1** Este ejercicio se refiere a los pasos que realiza un estudiante para levantarse e ir a la escuela. He aquí una sugerencia para los primeros pasos:

```
despertarse  
vestirse  
desayunar  
lavarse los dientes  
...
```

- Complete los pasos. Tenga en cuenta que no hay una secuencia de pasos ideal; los pasos varían de un individuo a otro.
  - El paso "lavarse los dientes" incluye una repetición, ya que representa una acción que hacemos una y otra vez. Identifique otro paso que contenga repetición.
  - Identifique un paso que contenga una selección.
  - Desglose uno de los pasos en varias acciones más pequeñas.
- 1.2** Suponga que recibe una enorme pila de papel que contiene 10,000 números, sin orden particular alguno. Explique por escrito qué proceso realizaría para encontrar el número más grande. Asegúrese de que su proceso sea claro y no tenga ambigüedades. Identifique cualquier selección o repetición que pudiera haber en su proceso.
- 1.3** Trate de escribir un conjunto de instrucciones precisas que permitan que un jugador gane una partida de gato (el juego de nueve casillas cuya meta es formar una línea de cruces o círculos). Si esto no es posible, trate de asegurar que un jugador no pierda.

### SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

---

**1.1** El principal error estriba en que la resta de deducciones por enfermedad se lleva a cabo demasiado tarde. Este paso debería ir antes de la impresión.

**1.2** Podríamos decir:

    siga dando masaje a su cabello hasta que esté completamente limpio.

o:

    continúe dando masaje a su cabello hasta obtener abundante espuma.

**1.3** El problema está en la palabra "o". Alguien que tenga 73 años también es mayor de 8 y, por lo tanto, podría subirse al juego.

    Para evitar la confusión podríamos sustituir la "o" por "y", de manera que la instrucción sea técnicamente correcta; no obstante, aun así la advertencia podría malentenderse. También podríamos escribir:

    sólo podrán subir a este juego las personas que tengan entre 8 y 70 años  
En este caso, sin embargo, es muy probable que tuviera que especificar si las personas de 8 y de 70 años también son admitidas en la atracción mecánica.



# El entorno de desarrollo de C#

**En este capítulo conoceremos cómo:**

- crear un proyecto con C#;
- manipular controles y sus propiedades en tiempo de diseño;
- ejecutar un programa;
- manejar un evento de clic de botón;
- mostrar un cuadro de mensaje;
- colocar texto en una etiqueta en tiempo de ejecución;
- localizar errores de compilación.

## ● Introducción

---

En este libro utilizaremos el acrónimo “IDE” (Integrated Development Environment) para referirnos al entorno de desarrollo integrado de C#. Al igual que los procesadores de palabras, programas que facilitan la creación de documentos, el IDE cuenta con herramientas para crear (desarrollar) programas. Todas las herramientas que requerimos para dicho propósito están integradas, por lo que podemos realizar la totalidad de nuestro trabajo dentro del IDE en vez de tener que usar otros programas. En otras palabras, el IDE nos proporciona un entorno de programación completo.

## ● Instalación y configuración

---

Para seguir los ejercicios de este libro es preciso que descargue Microsoft Visual C# 2008 Express Edition. Siga las instrucciones de instalación y registre el producto si el proceso te pide hacerlo.

A continuación le explicaremos cómo agrupar los programas que cree con C# en una carpeta específica.

Cuando se crea un programa, C# genera varios archivos y los coloca en una nueva carpeta. Al conjunto de archivos resultante se le conoce como “proyecto”. Es conveniente crear una carpeta de

nivel superior para guardar todos sus proyectos. La carpeta predeterminada que utiliza C# se llama **Projects** y se encuentra dentro de la carpeta **Mis documentos\Visual Studio 2008** en la unidad **C**. Casi todos los usuarios encuentran adecuada esta opción, pero si por algún motivo usted necesita modificar esa ubicación predeterminada, ponga en práctica estos pasos:

1. Haga clic en el menú **Herramientas**, ubicado en la parte superior de la pantalla. Seleccione **Opciones...** Asegúrese de que la opción **Mostrar todas las configuraciones** esté seleccionada.
2. Haga clic en la opción **Proyectos y soluciones** y después en el botón **...** que se encuentra a la derecha de la sección **Ubicación de proyectos**.
3. A continuación se abrirá la ventana **Ubicación del proyecto**, en la cual podrá elegir una carpeta existente o crear una nueva. Haga clic en el botón **Aceptar** cuando termine.

De aquí en adelante C# guardará todo el trabajo que usted realice en la carpeta de proyectos predeterminada o en la que haya seleccionado de manera específica. En cualquier caso, la primera vez que guarde un proyecto aparecerá un cuadro de diálogo con el botón **Examinar** (vea la Figura 2.9), por si necesita utilizar una carpeta distinta a la seleccionada.

## ● Cree su primer programa

El programa que crearemos a continuación desplegará el mensaje **Hola mundo** en la pantalla de su computadora. Independientemente de lo anterior, deberá seguir estos pasos generales para crear cualquier programa.

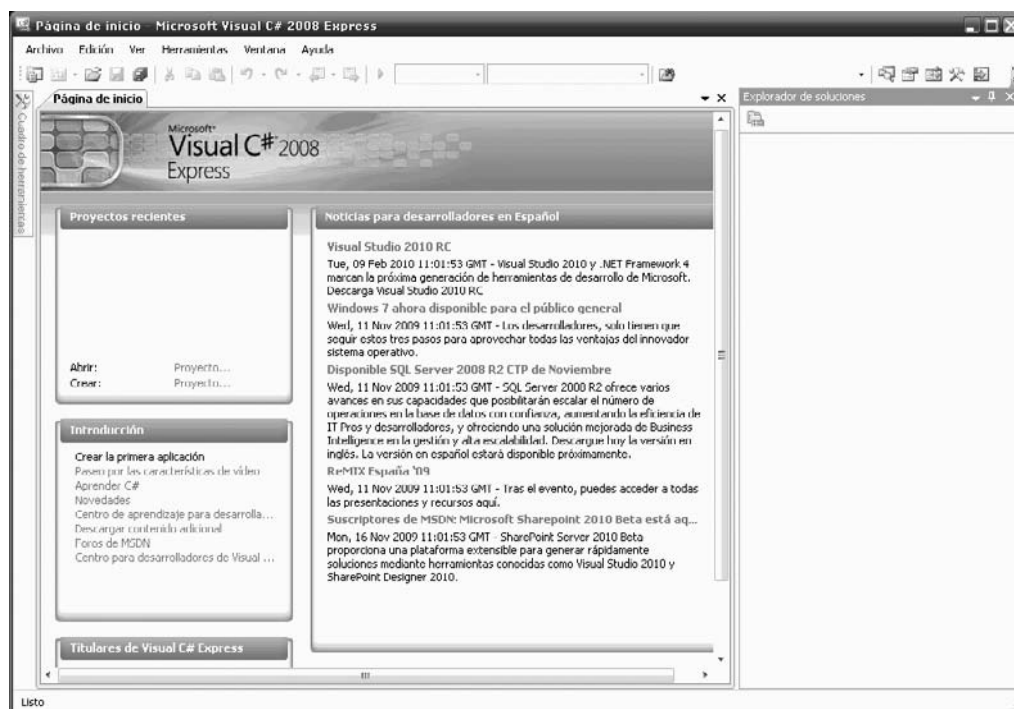


Figura 2.1 La Página de inicio.



Figura 2.2 La ventana Nuevo proyecto.

- Abra el IDE. A continuación aparecerá la Página de inicio, como se muestra en la Figura 2.1.
- Haga clic en el vínculo **Crear: Proyecto...** En la Figura 2.2 se muestra la ventana **Nuevo proyecto** que aparecerá en respuesta.
- Asegúrese de que esté seleccionada la plantilla **Aplicación de Windows Forms**. Elija un nombre para su proyecto, mismo que se convertirá también en la identificación de una carpeta. Le recomendamos utilizar sólo letras, dígitos y espacios. En nuestro caso utilizamos el nombre **Primer hola**. Haga clic en **Aceptar**. A continuación aparecerá un área de diseño similar (aunque no necesariamente idéntica) a la que se muestra en la Figura 2.4.
- Para facilitar su incursión en el lenguaje C#, es conveniente que el cuadro de herramientas esté siempre visible. Haga clic en el menú **Ver** y seleccione **Cuadro de herramientas**. Ahora haga clic en el símbolo de “chincheta” del cuadro de herramientas, como se muestra en la Figura 2.3; el cuadro de herramientas quedará fijo y abierto de manera permanente. Haga clic en **Todos los formularios Windows Forms** para ver la lista de herramientas disponibles. Ahora su pantalla se verá casi idéntica a la de la Figura 2.4.

Como se mencionó previamente, los pasos anteriores son comunes en la realización de cualquier proyecto. Realicemos ahora una tarea específica para este proyecto en particular.

Observe el formulario y el cuadro de herramientas ilustrados en la Figura 2.4. A continuación seleccionaremos un control del cuadro de herramientas, y lo colocaremos en el formulario. Éstos son los pasos a seguir:

- Localice el cuadro de herramientas y haga clic en el control **Label**.
- Desplace el puntero del ratón hacia el formulario. Haga clic y, sin soltar el botón, arrastre para crear una etiqueta como en la Figura 2.5.
- Ahora estableceremos algunas propiedades de la etiqueta: haga clic con el botón derecho del ratón sobre la etiqueta y seleccione **Propiedades**. Desplace el ratón hacia la lista Propiedades en la parte inferior derecha de la pantalla, como se muestra en la Figura 2.6.

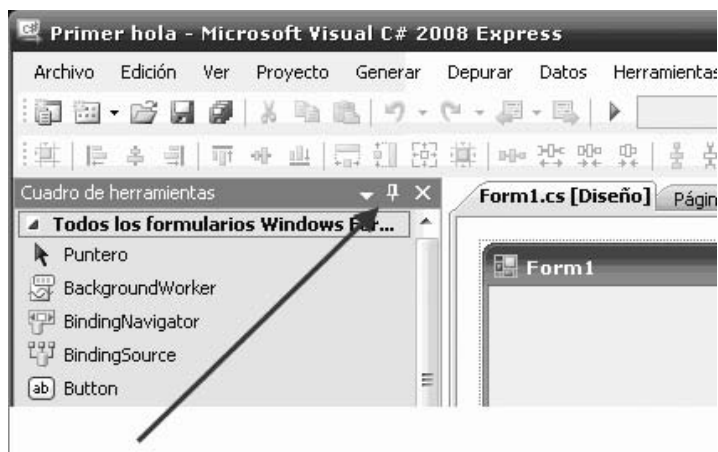


Figura 2.3 El cuadro de herramientas queda fijo y abierto.

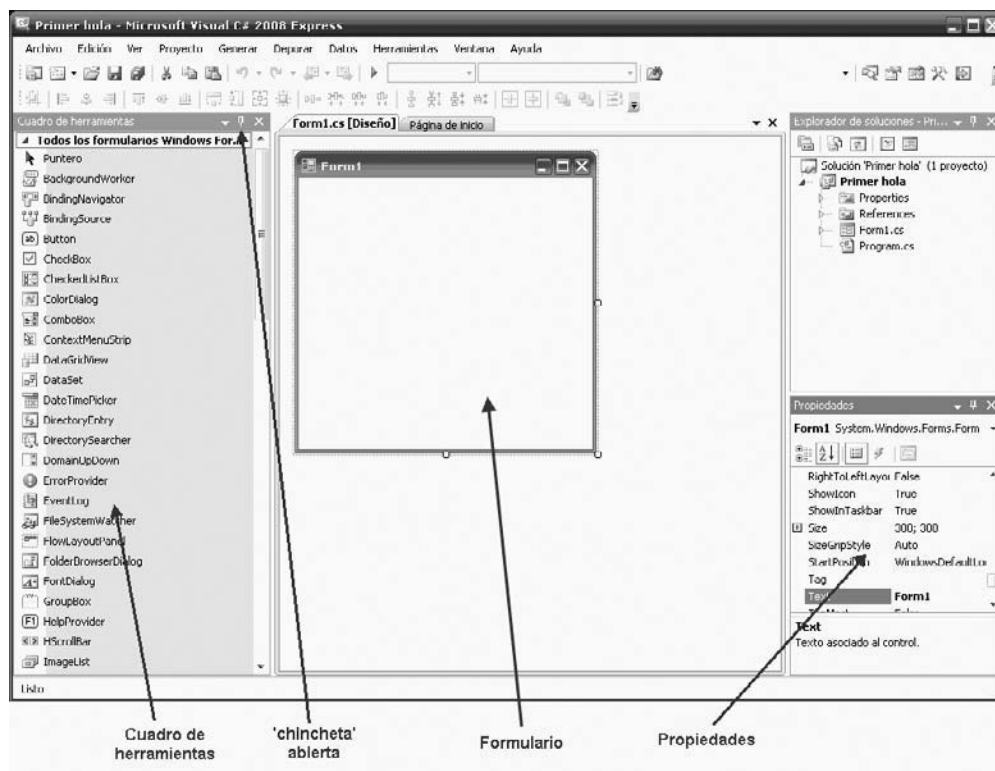


Figura 2.4 El IDE en tiempo de diseño.



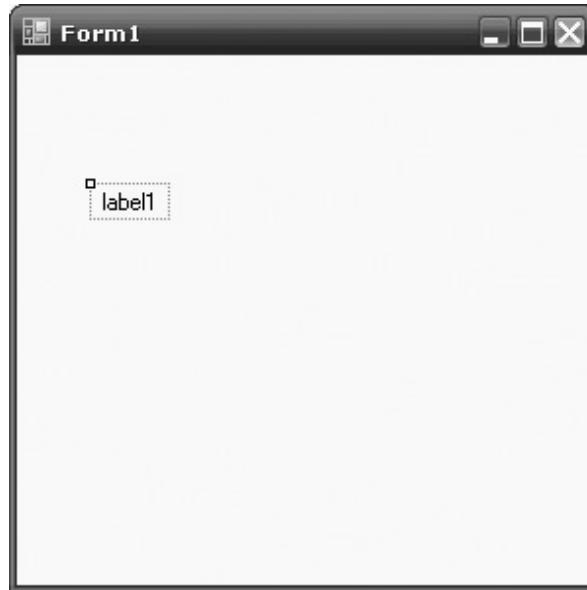


Figura 2.5 Se agregó una etiqueta al formulario.

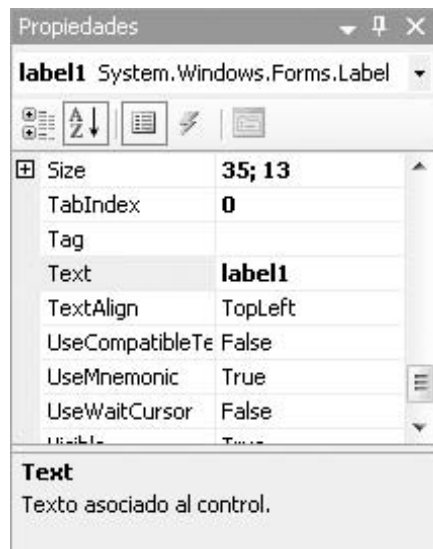


Figura 2.6 Propiedades de la etiqueta.



**Figura 2.7** Haga clic en la flecha para ejecutar el programa.

- Desplácese hasta la propiedad **Text** y sustituya el texto `label1` por **Hola Mundo**.
- Ejecutemos ahora el programa, haciendo clic en la flecha que se encuentra en la parte superior del IDE (Figura 2.7); se trata del botón Iniciar depuración.

Aparecerá una nueva ventana, como se muestra en la Figura 2.8. Es el programa que usted ha creado. Sólo muestra algo de texto, pero es una verdadera ventana en cuanto a que puede moverla, cambiar su tamaño y cerrarla al hacer clic en el icono **x** que se encuentra en la esquina superior derecha. Experimente con esta ventana, y después ciérrela.

Para guardar su programa y poder usarlo más adelante:

- Vaya al menú **Archivo** y seleccione la opción **Guardar todo** (en adelante, para indicar este tipo de acciones utilizaremos una forma abreviada como ésta: **Archivo | Guardar todo**).
- A continuación aparecerá la ventana **Guardar proyecto**, como se muestra en la Figura 2.9. Asegúrese de que la opción **Crear directorio para la solución** no esté marcada. Deje las demás opciones como están y haga clic en **Guardar**. Cuando vuelva a guardar el proyecto se utilizarán de manera automática las mismas opciones y ya no aparecerá la ventana **Guardar proyecto**.

Ahora puede utilizar el comando **Archivo | Salir** para cerrar el IDE.

La próxima vez que utilice C# el nombre de su proyecto aparecerá en la Página de inicio, de manera que podrá abrirlo con un solo clic, sin necesidad de repetir el trabajo que hicimos para configurar el proyecto.



**Figura 2.8** El programa **Primer hola** en ejecución.

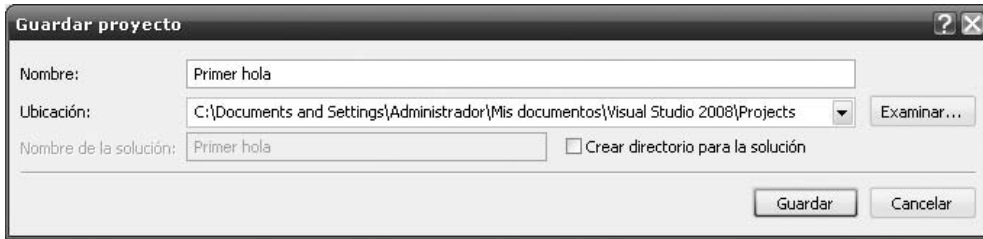


Figura 2.9 La ventana **Guardar proyecto**, con la opción **Crear directorio** desactivada.

## ● Los controles en tiempo de diseño

En nuestro programa **Primer Hola** colocamos una etiqueta en un formulario y modificamos el texto a desplegar. El propósito principal del ejercicio consistió en repasar los pasos involucrados en la creación de un proyecto. A continuación conoceremos algunos de los fundamentos de los controles y las propiedades.

¿Qué es un control? Es un “dispositivo” que aparece en la pantalla, ya sea para mostrar información, para permitir que el usuario interactúe con la aplicación, o ambas cosas. El mismo IDE de C# emplea muchos controles. Por ejemplo, en el ejercicio anterior usted utilizó los menús desplegables para guardar proyectos, y el botón **Aceptar** para confirmar acciones. En el caso de las aplicaciones para Windows, el cuadro de herramientas contiene cerca de 70 controles; parte de la tarea de programación implica seleccionar los controles apropiados, situarlos en un formulario y establecer sus propiedades. A diferencia del “tiempo de ejecución”, esta fase tiene relación con el uso de los controles y se denomina “tiempo de diseño”. Cuando el programa se ejecuta, el usuario interactúa con los controles. La labor del programador consiste en crear una interfaz gráfica de usuario (GUI) para facilitar dicha interacción. Veamos ahora cómo se pueden manipular los controles en tiempo de diseño.

- Es posible seleccionar un control del cuadro de herramientas y colocarlo en un formulario. La posición inicial que se le asigne no es importante, ya que podemos modificarla con facilidad.
- También es posible mover el control. Si coloca el puntero del ratón sobre un control, a su lado aparecerá una flecha con cuatro puntas, como se muestra en la Figura 2.10. En ese momento podrá arrastrar el control con el ratón.
- Se puede cambiar el tamaño del control. Al hacer clic en un control aparecen varios cuadros pequeños (controladores de tamaño) en sus bordes. Coloque el ratón sobre uno de estos cuadros pequeños y aparecerá una flecha con dos puntas, como se muestra en la Figura 2.11. Arrastre el borde o esquina para modificar el tamaño del rectángulo.

De hecho, el método para cambiar el tamaño depende del control en sí. Por ejemplo, el control tipo etiqueta tiene una propiedad llamada **AutoSize** (determinación automática de tamaño, de acuerdo

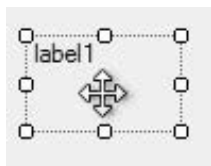


Figura 2.10 Los controles pueden moverse ...



Figura 2.11 ... y redimensionarse según convenga.

con las dimensiones del texto introducido) que establecemos como falsa (**False**), como se ilustra en la Figura 2.11. Si dejamos la propiedad **AutoSize** como verdadera (**True**), el ancho de la etiqueta se determinará con base al ancho del texto que vaya a mostrar, y la altura se determinará en función del tamaño de la fuente del texto. Otros controles permiten variar el ancho arrastrando los controladores con el ratón, pero no la altura (que se establece de acuerdo con las fuentes). Algunos controles —como los botones— permiten cambiar su tamaño en cualquier dirección. En general el tamaño de los controles depende del uso que se les vaya a dar, por lo que le recomendamos experimentar con ellos. Sin embargo, y dado que las etiquetas son tan comunes, no debemos olvidar su propiedad **AutoSize**.

A continuación examinaremos las propiedades. He aquí una analogía: los televisores tienen propiedades tales como el color de la carcasa, el tamaño de la pantalla, el canal que está mostrando en un momento dado, su precio y su marca.

De igual manera, cada control tiene un conjunto de propiedades que pueden ser ajustadas en tiempo de diseño para satisfacer nuestros requerimientos. Más adelante veremos cómo cambiar una propiedad en tiempo de ejecución.

Después de colocar un control en el formulario, para ver sus propiedades hay que hacer clic con el botón derecho sobre él y seleccionar **Propiedades**; de esta manera se desplegará la ventana de propiedades del control elegido. La columna izquierda contiene los nombres de las propiedades, y la columna derecha su valor actual. Para cambiar una propiedad debemos modificar el valor en la columna derecha. En algunas propiedades tal vez sea necesario seleccionar varias opciones adicionales, como al cambiar la configuración de los colores y las fuentes de texto. Algunas veces, cuando el rango de valores a elegir es muy amplio, se requiere abrir una ventana adicional.

Otro de los aspectos vitales de un control es su nombre. Éste no es en sí mismo una propiedad, pero por conveniencia se muestra en la lista de propiedades como (**Name**). Los paréntesis indican que en realidad no es una propiedad.

Cuando colocamos varias etiquetas en un formulario, el IDE selecciona sus nombres de la siguiente manera:

```
etiqueta1    etiqueta2    etiqueta3 ...
```

Estos nombres son aceptables por ahora, pero en los siguientes capítulos veremos que es mejor cambiar los nombres predeterminados de algunos controles por nombres más representativos. Para cambiar el nombre de un control es preciso modificar el texto que está a la derecha de (**Name**) en la lista de propiedades.

## PRÁCTICA DE AUTOEVALUACIÓN

**2.1** Coloque dos etiquetas en un formulario. Haga las siguientes modificaciones en sus propiedades. Después de realizar cada modificación ejecute el programa, observe el efecto y detenga la ejecución haciendo clic en el botón **x** en la esquina superior derecha de la ventana.

- Mueva las etiquetas.
- Establezca la propiedad **AutoSize** de una de las etiquetas en **True**.
- Altere sus propiedades **Text** de manera que la información desplegada sean su nombre y edad.
- Modifique sus fuentes utilizando la propiedad **Font**.
- Altere su color de fondo utilizando la propiedad **BackColor**.

**PRÁCTICA DE AUTOEVALUACIÓN**

**2.2** Seleccione el formulario completo. Realice las siguientes tareas, ejecutando el programa después de cada modificación.

- Cambie el tamaño del formulario.
- Altere su propiedad **Text**.
- Altere su propiedad **BackColor**.

## ● Los eventos y el control Button

El programa que creamos en el ejercicio anterior no es muy representativo, dado que siempre muestra las mismas palabras y el usuario no puede interactuar con él. Enseguida enriqueceremos este programa de manera que aparezca un mensaje de texto cuando el usuario haga clic en un botón. Éste es un ejemplo de cómo usar un *evento*.

Casi todos los eventos son puestos en acción por el usuario, y se generan cuando éste manipula un control de alguna forma en tiempo de ejecución. Cada control tiene varios eventos a los que puede responder, como el clic de un botón del ratón, un doble clic o la colocación del puntero del ratón sobre el control. Otros tipos de eventos no tienen su origen en el usuario; por ejemplo, la notificación de que una página Web ha terminado de descargarse.

En el programa que crearemos a continuación detectaremos un evento (el clic de un botón); después haremos que se despliegue un mensaje de texto en una etiqueta. He aquí la forma en que crearemos la interfaz de usuario:

- Cree un nuevo proyecto llamado **Botón hola**.
- Coloque una etiqueta y un botón en el formulario. La posición en que lo haga no es importante.
- Escriba **Haga clic aquí** en la propiedad **Text** del botón.
- Modifique la propiedad **Text** de la etiqueta, de forma que aparezca sin contenido.

Ejecute el programa, aunque todavía no está completo. Observe que puede hacer clic en el botón, y que éste cambia su aspecto ligeramente para confirmar que está siendo oprimido; no ocurre nada más. Cierre el formulario.

Veamos ahora cómo detectar el evento del clic. Haga doble clic en el botón dentro del formulario de diseño. A continuación se abrirá un nuevo panel de información, como el que se muestra en la Figura 2.12. Observe las fichas de la parte superior:

**Form1.cs**    **Página de inicio**    **Form1.cs [Diseño]**

Usted puede hacer clic en esas fichas para cambiar de panel. El panel **Form1.cs** muestra un programa de C#. A esto se le conoce como “texto del programa”, o “código” de C#. Modifiquemos este código utilizando el editor del IDE.

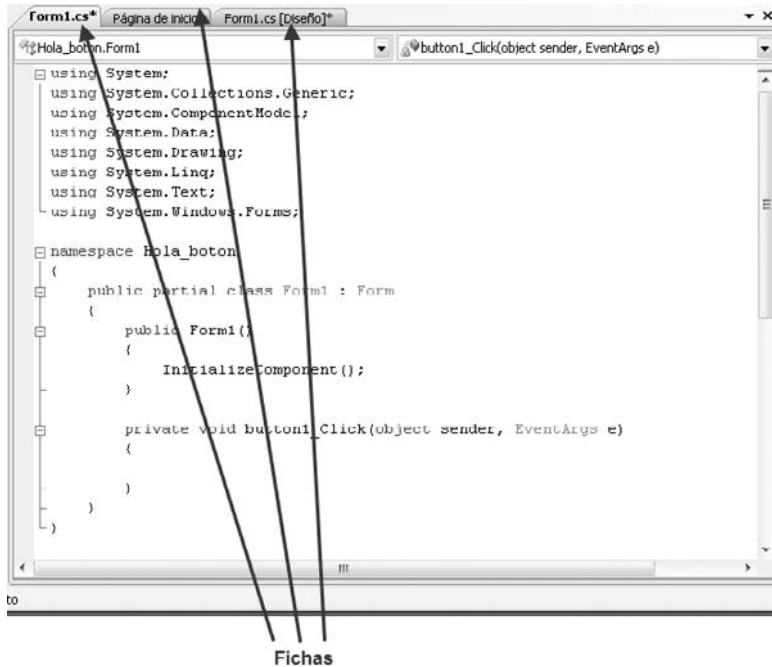


Figura 2.12 Código de C# en el panel de edición.

Desplácese por el código hasta que encuentre la siguiente sección:

```
private void button1_Click(object sender, EventArgs e)
{
}

```

A esta sección de código se le conoce como *método*. El nombre del método es `button1_Click`. Cuando el usuario haga clic sobre el botón `button1` en tiempo de ejecución, se llevará a cabo, o “ejecutará”, cualquier instrucción que coloquemos entre las dos líneas anteriores.

En este programa específico utilizaremos una instrucción para colocar el mensaje `Hola mundo` en el texto de la etiqueta `label1`. La instrucción para realizar esta acción es:

```
label1.Text = "Hola mundo";
```

Escriba la instrucción exactamente como se muestra aquí; hágalo entre las líneas `{` y `}`. En los siguientes capítulos explicaremos el significado exacto de líneas como la anterior (que implican una “instrucción de asignación”).

El siguiente paso es ejecutar el programa. Para ello hay dos posibilidades:

- Si el programa se ejecuta correctamente, al hacer clic en el botón observará que aparece el mensaje `Hola mundo` en la etiqueta.



**Figura 2.13** Acción errónea. Marque la casilla de verificación y haga clic en **No**.

- La otra posibilidad es que el programa no se ejecute debido a un error. En este caso C# mostrará un mensaje como el de la Figura 2.13. Marque la casilla de verificación de la opción **No volver a mostrar este cuadro de diálogo**, y haga clic en **No** para continuar. El mensaje no volverá a aparecer; en lo sucesivo la única evidencia del error será un subrayado en el texto del código.

De todas formas hay que corregir el error: revise el código, corrija cualesquiera errores de escritura, y ejecute el programa de nuevo. Más adelante hablaremos sobre los errores con más detalle.

He aquí las nuevas características de este programa:

- Puede responder al clic en un botón. Al proceso de colocar código de manera que se lleve a cabo la acción correspondiente cuando ocurra un evento se le conoce como “manejar” el evento.
- Modifica el valor de la propiedad de un control en tiempo de ejecución. Anteriormente sólo hacíamos esto en tiempo de diseño. Ésta es una parte muy importante de la programación, ya que a menudo tenemos que mostrar resultados colocándolos en cierta propiedad de un control.

#### PRÁCTICA DE AUTOEVALUACIÓN

**2.3** Coloque una segunda etiqueta en el formulario. Haga que muestre su nombre cuando se haga clic en el botón.

### ● Apertura de un proyecto existente

Para volver a abrir un proyecto creado con anterioridad, guarde y cierre el proyecto en progreso y vaya a la Página de inicio. En ella se muestran los proyectos en los que se ha trabajado más recientemente; para abrir uno basta con hacer clic en su nombre. Si el proyecto que busca no se encuentra en la lista, haga clic en el vínculo **Abrir: Proyecto...** y navegue hasta la carpeta apropiada. Dentro de la carpeta hay un archivo de tipo **.sln** (solución). Seleccione este archivo y abra el proyecto. Para ver el formulario vaya al Explorador de soluciones, ubicado en la parte superior derecha de la ventana, y haga doble clic en el nombre del formulario.

### ● Documentación de los valores de las propiedades

En este libro necesitamos proveerle los valores de las propiedades que emplearemos en los controles. Cuando las propiedades sean pocas podremos explicarlas en unos cuantos enunciados, pero si la can-



tividad es mayor utilizaremos una tabla. En general tenemos varios controles; cada control tiene varias propiedades y cada propiedad tiene un valor. Tenga en cuenta que hay una gran cantidad de propiedades para cada control, pero sólo listaremos aquellas que usted tenga que modificar. Las demás mantendrán su valor predeterminado. He aquí un ejemplo:

Control	Propiedad	Valor
label1	Text	Hola mundo
label1	BackColor	Red
button1	Text	Haga clic aquí

Los valores listados corresponden a los siguientes controles:

- `label1` cuyo texto es `Hola mundo`, y cuyo color de fondo es rojo;
- `button1` cuyo texto es `Haga clic aquí`.

Tenga cuidado al escribir los nombres de las propiedades. Hablaremos sobre esto con más detalle en capítulos posteriores; por ahora basta decir que debe hacerlo con letra mayúscula inicial, y sin espacios.

## ● Errores en los programas

Es común que ocurran errores en los programas; por fortuna, el IDE puede detectar algunos por nosotros. He aquí un ejemplo de error:

```
label1.Txtt = "Hola mundo"
```

Si usted escribe esta línea y ejecuta el programa, el IDE la subrayará. Al colocar el puntero del ratón sobre la parte subrayada aparecerá una explicación del error. Tenga en cuenta que la explicación podría ser difícil de comprender (incluso para un experto), o que tal vez el error se encuentre en otra parte del programa. Sin embargo, el primer paso deberá consistir siempre en inspeccionar el área subrayada para ver si hay errores de escritura. A medida que vaya aprendiendo más sobre el lenguaje C# mejorará su habilidad para detectar errores.

Una vez corregidos los errores subrayados hay que ejecutar el programa. De hecho, antes de que esto ocurra se activa otro programa llamado compilador, el cual realiza comprobaciones en el código para detectar, quizá, más errores. Sólo hasta que se hayan corregido todos los errores de compilación su programa podrá ejecutarse.

Los errores de compilación se muestran en la ventana **Lista de errores**, la cual se abre debajo del código. Al hacer doble clic en el error dentro de la ventana de resultados, el IDE nos llevará al punto del código donde se debe corregir el error.

Cualquiera que empiece a escribir programas enfrentará muchos errores de compilación. No se desanime; esto es parte del proceso de aprendizaje.

Tras corregir los errores de compilación el programa podrá ejecutarse. En este punto podríamos notar que lo hace de manera incorrecta, lo cual querría decir que hay un “error en tiempo de ejecución”, o *bug*. Dichos errores son más difíciles de corregir, y en consecuencia es necesario realizar una depuración. Hablaremos sobre esto en el capítulo 9.



## ● Funciones del editor

El editor no se limita a mostrar lo que escribe el programador; las siguientes son algunas de sus virtudes adicionales:

- Ofrece las operaciones estándar de cortar, copiar y pegar en el portapapeles, gracias a lo cual usted podrá copiar código de otros programas de Windows.
- Para escribir programas más grandes se requieren controles y eventos adicionales. Las listas desplegables que están en la parte superior del editor nos permiten seleccionar controles, además de un evento específico para dicho control. El editor se coloca de manera automática en el método apropiado.
- El editor desplegará el código de C# que se va tecleando de acuerdo con ciertas reglas. Por ejemplo, se insertarán espacios de manera automática alrededor del símbolo `=`.
- Algunas líneas necesitarán sangrías —desplazamiento del texto a la derecha—, para lo cual hay que insertar espacios adicionales en blanco. Por lo general esto se hace en intervalos de cuatro espacios. Como veremos en capítulos posteriores, el código de C# consiste de secciones dentro de otras secciones, y las sangrías nos ayudan a indicar en dónde inicia y termina cada sección.

Al proceso de aplicar sangrías a un programa también se le conoce como *darle formato*. El IDE puede hacerlo de manera automática a través del menú:

**Edición | Avanzado | Dar formato al documento** Es conveniente dar formato al código con frecuencia.

- Cada tipo de control cuenta con un conjunto de propiedades particular. Si usted escribe el nombre de un control seguido de un punto, como se muestra a continuación:

```
label1.
```

y espera un poco, el editor le proporcionará una lista de las propiedades de la etiqueta `label1`. Si esta herramienta no se encuentra configurada en su sistema, puede activarla seleccionando la opción:

**Herramientas | Opciones | Editor de texto | C#**

y marcando después la casilla de verificación **Lista de miembros automática**.

- Las secciones del código en las que no estemos trabajando pueden contraerse en una sola línea de código. Para ello hay que hacer clic en el pequeño símbolo “-” que está a la izquierda del editor. Para expandir una sección haga clic en el símbolo “+”. Encontrará estos símbolos al lado de líneas como la siguiente:

```
private void button1_Click(object sender, EventArgs e)
```

No es preciso que usted recuerde todas estas herramientas, pues siempre están al alcance de su mano para ayudarlo.

Una de las tareas relacionadas con la distribución del código que no se realizan de manera automática es la división de líneas extensas. Para asegurar que toda una línea esté visible en la pantalla podemos elegir un lugar adecuado para dividirla y oprimir la tecla **Enter**. El IDE aplicará una sangría de cuatro espacios a la segunda parte de la línea. He aquí un ejemplo:

```
private void button1_Click(object sender, EventArgs e)
```

Si presionamos la tecla **Enter** después de la coma, el código aparecerá así:

```
private void button1_Click(object sender,
    EventArgs e)
```

Muchas veces el IDE crea líneas muy extensas que, por supuesto, no deben contener errores, de manera que es mejor no dividirlas. Sin embargo, a medida que sea más experimentado en su manejo de C#, algunas de estas largas líneas serán de su propia creación, y resultará muy conveniente dividirlas de manera que pueda verse todo el texto a la vez. Con esto también se mejora la legibilidad del código impreso (conocido como listado).

**PRÁCTICA DE AUTOEVALUACIÓN**

2.4 Coloque una etiqueta y un botón en un formulario; después realice lo siguiente:

(a) En tiempo de diseño, haga doble clic en el botón e inserte esta línea:

```
label1.text = "Doug";
```

Observe el subrayado y el mensaje de error.

(b) Ejecute el programa. Observe el error en la Lista de errores y después haga doble clic en él. Corrijalo.

(c) Elimine los espacios a la izquierda de la línea y alrededor del signo =. Dé formato al documento y observe el efecto.

## ● El cuadro de mensajes

Anteriormente utilizamos el control tipo etiqueta para mostrar texto en la pantalla, pero también podemos usar un cuadro de mensajes para lograrlo. Este control no aparece en el cuadro de herramientas, debido a que no ocupa un espacio permanente en un formulario; sólo aparece cuando es requerido. El siguiente es un fragmento de código que muestra el mensaje `Hola mundo` dentro de un cuadro de mensajes al hacer clic en un botón:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hola mundo");
}
```

La Figura 2.14 muestra lo que ocurre al ejecutar el programa y hacer clic en el botón: se despliega el cuadro de mensajes, y debemos hacer clic en el botón **Aceptar** para que desaparezca. Esta característica implica que los cuadros de mensajes se utilizan para mostrar mensajes importantes que el usuario no puede ignorar.

Para utilizar un cuadro de mensajes escriba una línea como la anterior, pero utilice su propio mensaje dentro de las comillas dobles. En este momento no explicaremos el propósito de la instrucción `Show` ni por qué se requieren los paréntesis. Hablaremos sobre todo esto cuando estudiemos los métodos con más detalle.

**PRÁCTICA DE AUTOEVALUACIÓN**

2.5 Escriba un programa que tenga un formulario con dos botones. Al hacer clic en un botón deberá aparecer un cuadro con el mensaje `Hola mundo`. Al hacer clic en el otro deberá desplegarse un cuadro con el mensaje `Adiós, mundo cruel`.



Figura 2.14 Un cuadro de mensajes.

## ● Ayuda

El sistema de ayuda funciona con base en dos posibles fuentes: una local, ubicada en su propia computadora, y la otra a través de Internet. La fuente de Internet se actualiza con frecuencia; sin embargo, la versión local es suficiente cuando se empieza a trabajar con C#: En caso de que usted no cuente con una conexión a Internet, el programa utilizará automáticamente la ayuda local.

Si desea especificar de dónde debe provenir la ayuda, puede reconfigurar el sistema de la siguiente forma:

1. En la ventana principal de C#, seleccione **Ayuda | Buscar...**
2. En la nueva ventana de ayuda seleccione **Ayuda | Ayuda sobre la Ayuda**. A continuación aparecerán varias opciones sobre cuál fuente de ayuda tendrá mayor prioridad.

El sistema de ayuda de C# es extenso, pero si usted es nuevo en materia de programación tal vez la información que contiene le resulte complicada. Es más provechoso cuando se tiene algo de experiencia con C# y se requiere un detalle técnico preciso.

Las opciones más útiles del menú Ayuda son **Índice** y **Buscar**, ya que nos permiten escribir cierto texto para que el sistema localice las páginas que puedan servirnos. La diferencia es que **Índice** sólo busca en los títulos de las páginas, mientras que **Buscar** también examina el contenido de las mismas.

## Fundamentos de programación

- Los controles pueden colocarse en un formulario en tiempo de diseño.
- Es posible establecer las propiedades de los controles en tiempo de diseño.
- Los programas son capaces de modificar las propiedades en tiempo de ejecución.
- Cuando ocurre un evento (como hacer clic en un botón), el sistema de C# utiliza el método apropiado, pero es el programador quien debe colocar código dentro del método para manejar ese evento.

## Errores comunes de programación

- Olvidar dar por terminada la ejecución de su programa antes de tratar de modificar el formulario o el código.
- Confundir el formulario en tiempo de diseño con el formulario en tiempo de ejecución.

## Secretos de codificación

- En el código de C#, para referirnos a la propiedad de un control utilizamos su nombre seguido por un punto y por el nombre de la propiedad, como en:

```
label1.Text
```

- A una sección de código entre:

```
private void botonNumero_Click(Object sender, EventArgs e)
{
}

```

se le conoce como método.

- Los cuadros de mensaje no se colocan directamente en los formularios. Para hacer que aparezcan en pantalla debemos usar la siguiente instrucción:

```
MessageBox.Show("Aquí va el texto que usted desee");
```

### Nuevos elementos del lenguaje

Una introducción a las propiedades, métodos y eventos.

### Nuevas características del IDE

- Los programas están contenidos en proyectos.
- El IDE crea una carpeta que contiene los archivos necesarios para un proyecto.
- Es posible mover y cambiar el tamaño de los controles en los formularios.
- El cuadro de herramientas contiene una amplia diversidad de controles.
- Al hacer clic con el botón derecho del ratón sobre un control podemos seleccionar sus propiedades.
- Al hacer doble clic en un botón en tiempo de diseño se crean métodos para manejar eventos.

### Resumen

Parte de la tarea de programación implica colocar controles en formularios y establecer sus propiedades iniciales. El IDE de C# realiza esta tarea directamente, pero es necesario que practique con el IDE y lea sobre él.

### EJERCICIOS

- 2.1** Cree un nuevo proyecto llamado **Demo** y coloque tres botones en el formulario. Asigne los dígitos 1, 2 y 3, respectivamente, a sus propiedades de texto. Cree tres etiquetas y establezca sus propiedades de texto como **A**, **B** y **C**, respectivamente. Luego coloque el código necesario en los métodos apropiados de los botones, de manera que:
- (a) al hacer clic en `button1` el texto de todas las etiquetas cambie a **si**;
  - (b) al hacer clic en `button2` el texto de todos los botones cambie a **No**;
  - (c) al hacer clic en `button3` los valores de texto regresen a **A**, **B**, **C**.
- 2.2** En este ejercicio se debe utilizar la propiedad `visible` de un control, la cual puede establecerse como `true` o `false` (no utilice mayúsculas). Por ejemplo, el siguiente código hace invisible la etiqueta `label11`:

```
label11.Visible = false;
```

Escriba un programa con dos botones y una etiqueta. Al hacer clic en un botón la etiqueta deberá hacerse invisible, y al hacer clic en el otro deberá hacerse visible nuevamente.

- 2.3** Este programa implica el uso de la propiedad `Image` del control tipo etiqueta; esta propiedad provoca que el control muestre una imagen. Para establecerla es necesario utilizar un archivo de imagen. Seleccione cualquier imagen que tenga a mano en su equipo: hay muchas imágenes de muestra en la mayoría de las computadoras, y casi siempre sus nombres de archivo terminan en `.jpg` o `.bmp`. Escriba un programa con dos botones y una imagen en una etiqueta. Al hacer clic en un botón la imagen deberá desaparecer, y al hacer clic en el otro deberá desplegarse nuevamente.
- 2.4** Escriba un programa en el que primero muestre su nombre en un cuadro de mensaje, y después al hacer clic en un botón muestre su edad.
- 2.5** Este ejercicio involucra la creación de un editor de texto simple. Coloque un cuadro de texto en el formulario y cambie su tamaño, de manera que ocupe casi todo el espacio disponible. Establezca su propiedad `Multiline` en `true`, y su propiedad `ScrollBars` en `Both`. Ejecute el programa y escriba algo de texto en el cuadro. Observe que si hace clic con el botón derecho del ratón podrá realizar operaciones de cortar y pegar. Abra un procesador de palabras y pegue texto de su editor en el procesador de palabras y viceversa.
- 2.6** Este ejercicio implica utilizar el evento `MouseHover`, el cual ocurre cuando el usuario coloca el puntero del ratón sobre un control durante unos cuantos segundos. Para crear un método que maneje este evento coloque un botón en el formulario y, en la parte superior del panel del editor de texto, seleccione `button1` y `MouseHover`. A continuación se creará el método para manejar el evento. Escriba un programa que muestre un cuadro de mensajes que contenga el mensaje de texto sobre el botón cuando el puntero del ratón se pose encima del mismo.

## SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

- 2.1** Esta práctica requiere exploración: la manipulación de propiedades es una tarea práctica. Usted aprenderá a seleccionar y manipular los controles individualmente.
- 2.2** Observe que la propiedad `Text` afecta las palabras que se despliegan en el título del formulario.
- 2.3** Borre la propiedad `Text` de la etiqueta en tiempo de diseño (C# designará automáticamente este control como `label2`). Después agregue la siguiente línea:

```
label2.Text = "Mike";
```

Coloque esta línea justo debajo de la que muestra el texto `Hola mundo`. Ejecute el programa.

- 2.4** (a) Mantenga el puntero del ratón sobre la parte subrayada para ver el mensaje desplegable de error.
- (b) El cursor se colocará en la línea que contiene el error. Cambie la palabra `text` por `Text` (todas las propiedades deben empezar con mayúscula).
- (c) Al dar formato se añaden espacios alrededor del signo `=`.
- 2.5** Agregue el siguiente código a nuestro ejemplo del cuadro de mensajes:

```
private void button2_Click(object sender, EventArgs e)
{
    MessageBox.Show("Adiós mundo cruel");
}
```



# Introducción a los gráficos

**En este capítulo conoceremos cómo:**

- utilizar las herramientas de dibujo para formas simples;
- invocar métodos;
- pasar argumentos a los métodos;
- escribir programas como una secuencia de instrucciones;
- agregar comentarios a un programa.

## ● **Introducción**

---

El término “gráficos de computadora” evoca muchas posibilidades. Podríamos hablar de películas generadas por computadora, de sofisticados videojuegos, de entornos de realidad virtual, de una imagen estática de estilo fotográfico en un monitor, o de una imagen más simple, construida a partir de líneas. En este libro nos limitaremos a la visualización de imágenes estáticas construidas a partir de formas simples. Esta simpleza es intencional, ya que necesitamos concentrarnos en el uso de los objetos y los métodos, sin distraernos con los detalles gráficos.

## ● **Objetos, métodos, propiedades, clases: una analogía**

---

Algunas veces podemos explicar la programación orientada a objetos mediante una analogía. En este capítulo analizaremos el concepto de un equipo para dibujo de gráficos desde un punto de vista real, y después desde la perspectiva de la computación orientada a objetos. Tenga en cuenta que ésta es tan sólo una introducción, y que cubriremos este material con más detalle en los siguientes capítulos.

En el mundo real nuestro equipo de dibujo podría consistir en un montón de hojas de papel en blanco, algunos lápices y un conjunto de herramientas para dibujar formas (por ejemplo, una regla y una plantilla de figuras recortadas). Por supuesto, los lápices tendrían que ser adecuados para el papel que usemos: por ejemplo, si se tratara de hojas de acetato podríamos emplear lápices con tinta a base de aceite.

Tenga en cuenta que no es suficiente contar con papel y plantillas; lo que nos da la oportunidad de crear dibujos y gráficos es la *combinación* de estos elementos.

En el mundo computacional orientado a objetos, para comenzar a trabajar es necesario que solicitemos a C# un área de dibujo en blanco (tal como hacemos al seleccionar un “nuevo” documento antes de empezar a escribir en un procesador de palabras). Esta área de dibujo incluye un conjunto de *métodos* (funciones, operaciones) para dibujar formas. La idea de una hoja de papel que no haga nada va en contra de la metodología orientada a objetos. Para expresarlo de otro modo: en el estilo de objetos de C#, obtenemos una hoja de papel ‘inteligente’ que incluye un conjunto de herramientas.

¿Cuántos proyectos de dibujo podemos crear? En términos de computación no hay límites específicos. Por ejemplo, en un procesador de palabras podemos crear tantas ventanas de nuevos documentos como sea necesario con sólo hacer clic en el botón “Nuevo”. De hecho, en C# se utiliza la palabra **new** para proveer al programador objetos recién creados con los que pueda trabajar. Al utilizar **new** también debemos especificar qué tipo de nuevo objeto requerimos. En otras palabras, seleccionamos la clase del objeto. C# tiene una extensa colección de clases listas para usar (como botones, etiquetas, formularios, etc.).

Veamos ahora un ejemplo de cómo luce el código real. El código para dibujar un rectángulo sería algo así:

```
paper.DrawRectangle(lápiz a usar, posición del rectángulo, etc.)
```

Por ahora ignoraremos los detalles sobre el color y la posición del rectángulo. Lo importante es que **papel** es un objeto. Para dibujar sobre él utilizamos uno de sus métodos. En este caso elegimos el método **DrawRectangle** (dibujar rectángulo). Al proceso de utilizar un método se le conoce como “llamar” o “invocar” el método. Para invocar el método de un objeto utilizamos la notación “punto”, es decir, colocamos un “.” entre el objeto y el método que estamos invocando.

Además de métodos, los objetos también pueden tener *propiedades*; sin embargo, es imposible invocar propiedades, ya que éstas no realizan tareas por nosotros, sino que nos permiten ver o modificar el *estado* actual (valores) de un objeto. Por ejemplo, la propiedad **Text** de un botón contiene el mensaje que éste muestra en pantalla. Podemos establecer esta propiedad en tiempo de diseño, y de hecho también en tiempo de ejecución si así lo deseamos.

En el siguiente código se invocarán los métodos de la clase **Graphics** que nos proporciona C# (como **DrawRectangle**, entre otros). Nuestra área de dibujo (a la que llamaremos **papel**) será en efecto un control de cuadro de imagen (**PictureBox**), disponible a través del cuadro de herramientas.

También crearemos un nuevo objeto tipo lápiz (Pen), y estableceremos su color.

## ● Nuestro primer dibujo

A continuación crearemos un programa que muestra dos rectángulos en un cuadro de imagen cuando se hace clic en el botón, como puede verse en la Figura 3.1. Todas las instrucciones están incluidas en un método. He aquí el listado del código:

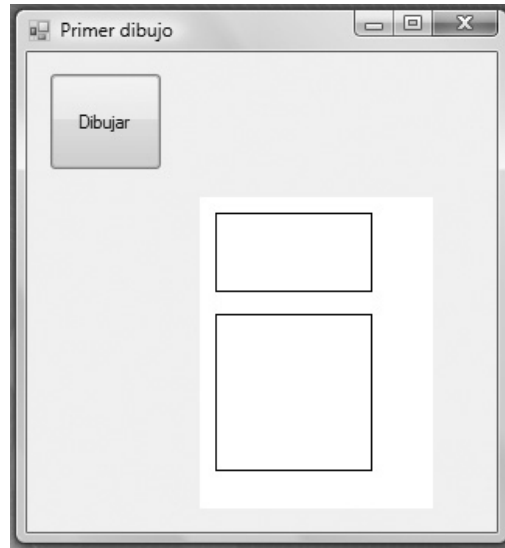


Figura 3.1 El programa Primer dibujo.

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    Pen lápiz = new Pen(Color.Black);

    papel.DrawRectangle(lápiz, 10, 10, 100, 50);
    papel.DrawRectangle(lápiz, 10, 75, 100, 100);
}
```

### ● Creación del programa

Para crear este programa utilice el IDE como se explica en el capítulo 2. Básicamente los pasos son:

- Entrar al IDE de C#.
- Crear un nuevo proyecto Aplicación de Windows Forms, llamado (por ejemplo) Primer dibujo.

Ahora debemos colocar controles en el formulario, de la siguiente manera:

- Coloque un botón y un cuadro de imagen (`PictureBox`) en el formulario. La posición exacta de los controles no es importante, pero puede utilizar la imagen de la Figura 3.1 como guía. Haga clic en el botón y cambie su propiedad `Text` a `Dibujar`.
- Haga clic en el cuadro de imagen y cambie su propiedad `Size` a `150, 200`.



- Modifique la propiedad `BackColor` del cuadro de imagen a un color adecuado, como el blanco (White). Para ello abra la lista desplegable que está a la derecha de la propiedad `BackColor` y seleccione **Personalizado**. A continuación aparecerán muestras de colores; haga clic en el color blanco.
- Si lo desea puede cambiar el texto utilizado en el título del formulario; para ello basta con hacer clic en él y cambiar su propiedad `Text` por un título representativo, como **Primer dibujo**. De hecho puede seleccionar cualquier título que desee; no es preciso que éste coincida con el nombre del proyecto.

El siguiente es un resumen de la configuración de los controles:

Control	Propiedad	Valor
button1	Text	Dibujar
pictureBox1	BackColor	(Personalizado) White
pictureBox1	Size	150, 200
Form1	Text	Primer dibujo

La etapa final de la creación del programa consiste en hacer doble clic en el botón e insertar las instrucciones para dibujar. Todas las instrucciones van dentro del método `button1_Click1`, como se indica en el código anterior.

Ejecute el programa; haga clic en el botón **Dibujar** para ver los dos rectángulos. Si se producen errores de compilación corrija la escritura e inténtelo de nuevo. Este tipo de equivocaciones es muy común, así que no debe preocuparse.

Para continuar examinaremos los detalles relacionados con el dibujo de formas.

## ● El sistema de coordenadas de gráficos

Los gráficos de C# se basan en píxeles. Los píxeles son pequeños puntos luminosos de la pantalla cuyo color puede cambiarse de manera específica. Cada píxel se identifica mediante un par de números (sus coordenadas), empezando desde cero:

- el primer número corresponde a su posición horizontal; a menudo se le denomina  $x$  en matemáticas, y también en la documentación de C#. Este valor se incrementa de izquierda a derecha;
- el segundo señala su posición vertical, o  $y$ ; este valor se incrementa de arriba hacia abajo.

Cuando colocamos un objeto visual en la pantalla establecemos su posición  $x$  y  $y$ . La Figura 3.2 muestra una forma con un tamaño de **400** por **200**, con un control `PictureBox` situado en las coordenadas **200, 100**. Sin embargo, al dibujar en el cuadro de imagen (`PictureBox`) consideramos *su* esquina superior izquierda como el punto cero de las coordenadas horizontal y vertical. En otras palabras, dibujamos en relación con la esquina superior izquierda del cuadro de imagen, y no de acuerdo con la esquina superior izquierda del formulario. Esto significa que si cambiamos la posición del cuadro de imagen no se verán afectados los dibujos que éste contenga. Utilizamos este sistema cuando pedimos a C# que dibuje formas simples.

El tamaño de los dibujos depende de la calidad de la pantalla del equipo en donde se ejecute el programa, y de la configuración gráfica del sistema. Los gráficos de alta calidad tienen más píxeles (aunque de menor tamaño), por lo que sus dibujos serán más pequeños.

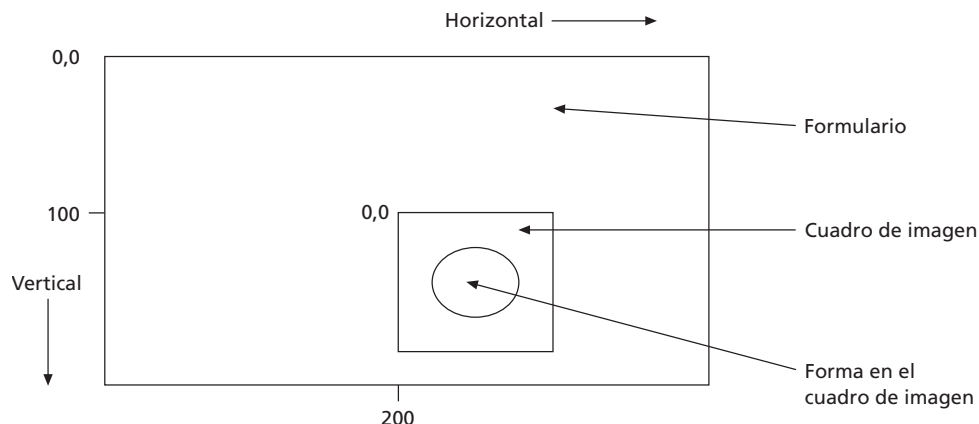


Figura 3.2 El sistema de coordenadas de píxeles en C#.

## ● Explicación del programa

No explicaremos aquí todos los detalles de cada línea, sólo las generalidades. Pero no se preocupe; en capítulos posteriores veremos todos esos detalles.

Nuestro código para dibujar está dentro de un método llamado `button1_Click`. El código se divide en dos fases: primero preparamos las herramientas de dibujo, y después dibujamos. Ésta es la primera parte:

```
Graphics papel;
papel = pictureBox1.CreateGraphics();
Pen lápiz = new Pen(Color.Black);
```

La buena noticia es que las líneas que acabamos de mostrarle serán las mismas que utilizaremos en casi todos nuestros programas de dibujo. A continuación le explicaremos brevemente su funcionamiento, aunque no es imprescindible que comprenda todos los detalles en este momento.

- En la línea 1 elegimos el nombre de nuestra área de dibujo (`papel`). También debemos declarar qué tipo de objeto es nuestro papel; es decir, su clase (en este ejemplo es un elemento tipo `Graphics` y no `Button`).
- En la segunda línea enlazamos nuestra área de dibujo con el cuadro de imagen que se colocó en el formulario.
- En la línea 3 elegimos un nombre para nuestro lápiz. La convención es que los nombres de las clases empiecen con mayúscula, y C# tiene una clase llamada `Pen`. En este caso escogimos el nombre `lápiz`, pero pudimos haber elegido `marcador`, por ejemplo. Si hubiéramos seleccionado el nombre `pen` para nuestro lápiz, C# sabría diferenciar entre los objetos `pen` y `Pen` debido a que la primera letra del segundo objeto (en realidad, una clase) es mayúscula.

Con esto terminamos nuestra preparación para dibujar. Le recomendamos utilizar estas tres líneas en todos los programas de este capítulo. Ahora estamos listos para dibujar algunas formas en nuestro cuadro de imagen.

Para dibujar las formas hay que invocar (o llamar) los métodos de dibujo de C#. Éste es el código para hacerlo:

```
papel.DrawRectangle(lápiz, 10, 10, 100, 50);
papel.DrawRectangle(lápiz, 10, 75, 100, 100);
```

El método `DrawRectangle` es uno de varios que proporciona la biblioteca del sistema C#. Observe el uso de las mayúsculas, mismo al que debemos apegarnos. Cada una de las instrucciones anteriores es una invocación (o llamada) al método, en la cual le pedimos que lleve a cabo la tarea de dibujar un rectángulo. También se le dice método debido a que es un procedimiento para (o medio de) realizar cierta acción.

Cuando utilizamos el método `DrawRectangle` es necesario que le proporcionemos un instrumento para realizar el dibujo (el lápiz) y los valores para fijar su posición y tamaño. Tenemos que suministrarle estos valores en el orden correcto:

- un objeto `Pen`;
- el valor horizontal de la esquina superior izquierda del rectángulo;
- el valor vertical de la esquina superior izquierda del rectángulo;
- el ancho del rectángulo;
- la altura del rectángulo.

En C# estos elementos reciben la denominación de argumentos. Otros lenguajes utilizan también el término “parámetro”. En este caso los argumentos constituyen entradas de información (*inputs*) para el método `DrawRectangle`. Los argumentos deben ir encerrados entre paréntesis y separados por comas. Este método específico tiene varias versiones distintas, con diferentes números y tipos de argumentos. El que utilizamos en este programa tiene cinco argumentos: un objeto `Pen` seguido de cuatro números enteros. Si tratamos de utilizar el número incorrecto de argumentos, o el tipo equivocado de éstos, obtendremos un mensaje de error del compilador. Para evitarlo necesitamos asegurarnos de:

- proveer el número correcto de argumentos;
- proporcionar el tipo correcto de argumentos;
- ordenarlos de la forma correcta.

Algunos métodos no requieren argumentos. En este caso también debemos utilizar paréntesis, como en este ejemplo:

```
pictureBox1.CreateGraphics();
```

En nuestro último programa invocamos métodos preexistentes, pero con la ayuda de C# hemos escrito nuestro propio método. C# le asignó el nombre `Button1_Click`. No necesitamos invocarlo de manera explícita, ya que C# se encarga de ello cuando el usuario hace clic en el botón `Button1`. La tarea del programa es invocar el método `DrawRectangle` dos veces. En el capítulo 5 hablaremos con más detalle sobre cómo escribir nuestros propios métodos.

Por último, observe el uso del signo “;” al final de algunas líneas. Esto se debe a que en C# debe aparecer un signo de punto y coma al final de cada “instrucción”, pero ¿qué es una instrucción? La respuesta a esta pregunta es muy importante. No obstante, como veremos al crear programas posteriores, no siempre aparece un punto y coma al final de *cada* línea. Por ahora sólo le pedimos que base sus programas en nuestros ejemplos para evitar confusiones. Sin embargo, podemos decir que invocar un método es de hecho una instrucción, por lo cual se requiere un punto y coma al final de la línea en donde se haga la llamada.

## ● Métodos para dibujar

---

Además de rectángulos, C# nos proporciona herramientas para dibujar diversas formas. Algunas de las más simples son:

- líneas;
- elipses (óvalos) y círculos;
- rectángulos, elipses y círculos rellenos;
- imágenes que tengamos almacenadas en archivos.

Además podemos cambiar el color de los lápices que utilizamos para dibujar, y usar “pinceles” del color que elijamos para rellenar formas.

A continuación se listan los argumentos para cada método, y se ofrece un programa de ejemplo (llamado **Algunas Formas**) que los utiliza.

### **DrawRectangle**

- un objeto lápiz (**Pen**);
- el valor horizontal de la esquina superior izquierda del rectángulo;
- el valor vertical de la esquina superior izquierda del rectángulo;
- el ancho del rectángulo;
- la altura del rectángulo.

### **DrawLine**

Tenga en cuenta que este método no utiliza el concepto de un rectángulo circundante. Los argumentos son:

- un objeto lápiz (**Pen**);
- el valor horizontal del inicio de la línea;
- el valor vertical del inicio de la línea;
- el valor horizontal del final de la línea;
- el valor vertical del final de la línea.

### **DrawEllipse**

Imagine la elipse (un óvalo) ajustada dentro de un rectángulo. Los argumentos a proporcionar son:

- un objeto lápiz (**Pen**);
- el valor horizontal de la esquina superior izquierda del rectángulo;
- el valor vertical de la esquina superior izquierda del rectángulo;
- el ancho del rectángulo;
- la altura del rectángulo.

Para producir formas rellenas contamos con los siguientes métodos:

## FillRectangle

Sus argumentos de coordenadas son casi idénticos a los del método `Draw` equivalente. La principal diferencia es que el primer argumento debe ser un objeto pincel (`Brush`) en vez de lápiz (`Pen`). Los pinceles pueden utilizar todo un rango de colores y texturas. Aquí sólo mostramos la versión más simple, sin textura:

```
SolidBrush miPincel = new SolidBrush(Color.Black);
papel.FillRectangle(miPincel, 10, 10, 90, 90);
```

## FillEllipse

Este método se utiliza de manera similar a `DrawEllipse`, sólo que se emplea un objeto pincel en lugar de un lápiz.

## DrawImage

Este método es muy distinto, ya que no utiliza formas preestablecidas. En vez de ello puede utilizarse para mostrar imágenes que se hayan almacenado en archivos, y que provengan de un programa para dibujar, de un escáner o de una cámara fotográfica. Para utilizar `DrawImage` primero debemos crear un objeto mapa de bits (`Bitmap`), proporcionando el nombre de un archivo que contenga una imagen (una pequeña observación: el carácter `@` tiene que ir antes del nombre del archivo, debido a que el carácter `\` en el nombre de éste tiene un significado especial dentro de las comillas. La `@` cancela este significado especial). Después utilizamos `DrawImage` especificando el mapa de bits, la posición de la imagen y el tamaño del rectángulo que la rodea. La imagen se recorta si es demasiado grande para caber en el rectángulo. El programa “Algunas formas” con el que trabajaremos a continuación nos enseña a crear el objeto mapa de bits, al que llamaremos `ima`. Para ello es necesario crear la imagen en un paquete de dibujo (puede emplear uno tan sencillo como el Paint de Windows), y guardarla como `demoimagen.jpeg`. También es posible trabajar con los tipos de archivo `gif` y `bmp`.

El orden de los argumentos para `DrawImage` es:

- un objeto mapa de bits que contiene una imagen de un archivo;
- el valor horizontal de la esquina superior izquierda del rectángulo;
- el valor vertical de la esquina superior izquierda del rectángulo;
- el ancho del rectángulo;
- la altura del rectángulo.

## ● Colores

Es posible crear tantos lápices y pinceles como se desee, cada uno con su propio color. En C# hay alrededor de 150 colores con nombre. A continuación listamos los colores principales, junto con algunos no tan utilizados:

Black	Violet	Blue
Indigo	Green	Yellow
Orange	Red	Gray
Purple	White	Firebrick
LemonChiffon	Maroon	OliveDrab

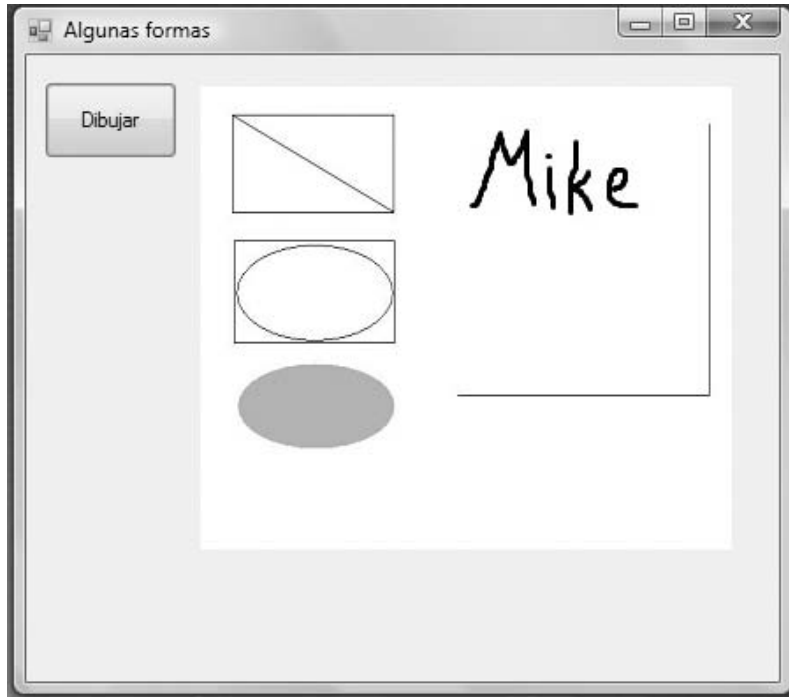


Figura 3.3 El programa Algunas formas.

Utilizamos los colores cuando creamos lápices y pinceles.

El siguiente es el código de un programa llamado Algunas formas, el cual dibuja una variedad de formas. La Figura 3.3 muestra el resultado que se obtiene al ejecutar este programa.

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    Bitmap ima = new Bitmap(@"c:\mike\vbbook\demoimagen.jpeg");
    Pen lápiz = new Pen(Color.Black);
    SolidBrush pincelRelleno = new SolidBrush(Color.Gray);
    papel.DrawRectangle(lápiz, 10, 10, 100, 50);
    papel.DrawLine(lápiz, 10, 10, 110, 60);
    papel.DrawRectangle(lápiz, 10, 80, 100, 50);
    papel.DrawEllipse(lápiz, 10, 80, 100, 50);
    papel.FillEllipse(pincelRelleno, 10, 150, 100, 50);
    papel.DrawRectangle(lápiz, 130, 10, 150, 150);
    papel.DrawImage(ima, 130, 10, 150, 150);
}
```

Cree el programa de la misma forma que el programa Primer dibujo, pero aumente el tamaño del cuadro de imagen a 300, 300.

#### PRÁCTICA DE AUTOEVALUACIÓN

**3.1** Escriba varias instrucciones de C# que produzcan un círculo relleno de color negro, con un radio de 50 píxeles y que se encuentre a una distancia de 10 píxeles desde la esquina superior izquierda de un cuadro de imagen.

### ● El concepto de secuencia y las instrucciones

Cuando tenemos varias instrucciones en un programa, éstas se ejecutan (se llevan a cabo, son obedecidas o realizadas...) en secuencia, de arriba abajo (a menos que especifiquemos lo contrario mediante el uso de los conceptos de selección y repetición que veremos en capítulos posteriores). Sin embargo, usted no podrá detectar esta acción, debido a la velocidad de la computadora.

En general los programas de C# están compuestos por una serie de *instrucciones*. Hay muchos tipos de instrucciones, como la invocación a un método o una asignación. Algunas instrucciones ocupan una sola línea, mientras que otras (como `if` y `while`, las cuales veremos más adelante) necesitan escribirse de manera que se distribuyan en varias líneas.

#### PRÁCTICA DE AUTOEVALUACIÓN

**3.2** Escriba y ejecute un programa que dibuje la forma de una "T" grande en la pantalla.

### ● Adición de significado mediante el uso de comentarios

¿Qué hacen las siguientes instrucciones?

```
papel.DrawLine(lápiz, 20, 80, 70, 10);
papel.DrawLine(lápiz, 70, 10, 120, 80);
papel.DrawLine(lápiz, 20, 80, 120, 80);
```

El significado no es obvio de inmediato, y probablemente usted haya tratado de averiguarlo utilizando lápiz y papel. La respuesta es que tales instrucciones dibujan un triángulo con una base horizontal, pero es difícil deducirlo de un solo vistazo.

En C# podemos agregar comentarios (un tipo de anotación) a las instrucciones mediante el uso de los caracteres `//` o utilizando `/* ... */`. El método más simple consiste en emplear `//` antes de nuestro comentario, como en el siguiente ejemplo:

```
// dibuja un triángulo
papel.DrawLine(lápiz, 20, 80, 70, 10);
papel.DrawLine(lápiz, 70, 10, 120, 80);
papel.DrawLine(lápiz, 20, 80, 120, 80);
```

Los comentarios pueden contener lo que se desee, no hay reglas definidas. Es responsabilidad de usted utilizarlos para expresar cierto significado.

También podemos colocar comentarios al final de una línea, como en el siguiente ejemplo:

```
// dibuja un triángulo
papel.DrawLine(lápiz, 20, 80, 70, 10);
papel.DrawLine(lápiz, 70, 10, 120, 80);
papel.DrawLine(lápiz, 20, 80, 120, 80); // dibuja la base
```

La segunda forma en que podemos usar los comentarios estriba en encerrar el texto entre los caracteres `/*` y `*/`, como en el siguiente ejemplo:

```
/*
papel.DrawRectangle(lápiz, 5, 20, 110, 120);
papel.DrawRectangle(lápiz, 5, 20, 70, 80);
*/
papel.DrawRectangle(lápiz, 5, 20, 10, 80);
```

En este último ejemplo sólo se ejecuta la última línea, ya que las dos primeras se consideran comentarios.

También podemos utilizar los caracteres `/*` y `*/` para convertir una sección del código en comentarios, mientras trabajamos en una versión alternativa.

No es conveniente abusar de los comentarios, y tampoco es recomendable comentar cada una de las líneas de un programa, ya que a menudo se corre el riesgo de duplicar información. El siguiente es un ejemplo del mal empleo de un comentario:

```
Pen lápiz = new Pen(Color.Black); // crea un lápiz negro
```

Aquí la instrucción indica con claridad lo que hace, sin que haya necesidad de un comentario. Use los comentarios para declarar las generalidades de una sección del programa, en vez de hacerlo para recalcar los detalles de cada una de las instrucciones implicadas.

#### **PRÁCTICAS DE AUTOEVALUACIÓN**

##### **3.3** Proporcione un comentario adecuado para estas líneas:

```
papel.DrawLine(miLápiz, 0, 0, 100, 100);
papel.DrawLine(miLápiz, 100, 0, 0, 100);
```

##### **3.4** ¿Qué despliega en pantalla el siguiente programa?

```
/*
papel.DrawRectangle(lápiz, 5, 20, 10, 80); */
papel.DrawRectangle(lápiz, 5, 5, 50, /* 70 */50);
```

## **Fundamentos de programación**

- C# tiene una gran colección de métodos que podemos llamar en nuestros programas.
- Los argumentos que pasamos a los métodos para dibujar gráficos tienen el efecto de controlar las formas que se dibujan.



## Errores comunes de programación

- Tenga cuidado con la puntuación, la ortografía y el uso de mayúsculas en los nombres de los métodos. Las comas y los paréntesis deben escribirse exactamente como se muestra en los ejemplos.

## Secretos de codificación

El orden y tipo de los argumentos deben ser correctos para cada método.

## Nuevos elementos del lenguaje

- ( ) para encerrar los argumentos.
- El uso del nombre de una clase al declarar elementos.
- El uso de `new` para crear nuevos objetos.
- El uso de la notación “punto” para invocar métodos de una clase.
- El uso de los caracteres `//` antes de los comentarios, y de los caracteres `/*` y `*/` para encerrar comentarios.
- El uso de `;` para terminar una instrucción.

## Nuevas características del IDE

No se abordó ninguna nueva característica en este capítulo.

## Resumen

- Las instrucciones son obedecidas o llevadas a cabo en secuencia, de arriba hacia abajo (a menos que solicitemos lo contrario).
- C# cuenta con un conjunto de métodos para “dibujar”, los cuales podemos invocar en nuestros programas para mostrar gráficos.
- La colocación de gráficos se basa en coordenadas de píxeles.
- Es posible pasar valores como argumentos a los métodos.

## EJERCICIOS

Para realizar los siguientes ejercicios le recomendamos hacer borradores y cálculos antes de escribir el programa. Puede utilizar el mismo proyecto para cada ejercicio, un cuadro de imagen para dibujar y un evento de botón para iniciar el dibujo.

**3.1** Escriba un programa que dibuje un triángulo rectángulo. Seleccione un tamaño apropiado.

- 3.2** Escriba un programa que dibuje un tablero de gato (círculos y cruces) vacío, hecho a partir de líneas.
- 3.3** Diseñe una casa simple, y después escriba un programa para dibujarla.
- 3.4** Las siguientes cifras corresponden a las precipitaciones pluviales anuales en el país de Xanadú, mismas que queremos mostrar en forma gráfica:
- |      |     |    |
|------|-----|----|
| 1998 | 150 | cm |
| 1999 | 175 | cm |
| 2000 | 120 | cm |
| 2001 | 130 | cm |
- (a) Muestre los datos como una serie de líneas horizontales.
- (b) En vez de líneas utilice rectángulos rellenos.
- 3.5** Escriba un programa que despliegue una diana de tiro al blanco con círculos concéntricos de distintos colores.
- 3.6** Escriba un programa que despliegue un rostro sencillo. El contorno del rostro, los ojos, orejas, nariz y boca pueden formarse con elipses.
- 3.7** Cree una imagen en un paquete de dibujo, y guárdela como archivo `bmp`, `gif` o `jpeg` (`jpg`). Escriba un programa que muestre esta imagen en pantalla y utilice `DrawRectangle` para dibujar un marco apropiado alrededor de la misma.

#### SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

---

- 3.1** Imagine que el círculo está ajustado dentro de un cuadrado, cada uno de cuyos lados mide 100 píxeles de largo:

```
SolidBrush pincelRelleno = new SolidBrush(Color.Black);  
papel.FillEllipse(pincelRelleno, 10, 10, 100, 100);
```

- 3.2**
- ```
papel.DrawLine(lápiz, 20, 20, 120, 20);  
papel.DrawLine(lápiz, 70, 20, 70, 120);
```

- 3.3** Las instrucciones dibujan una gran “X” en el cuadro de imagen, por lo que un comentario adecuado podría ser:

```
//dibuja una 'X' en la esquina superior izquierda
```

- 3.4** El programa dibuja un solo rectángulo de 50 por 50 píxeles. La primera línea se toma como un comentario, al igual que el 70 de la última línea.

# 4

## Variables y cálculos

**En este capítulo conoceremos:**

- los tipos de variables numéricas;
- cómo declarar variables;
- la instrucción de asignación;
- los operadores aritméticos;
- el uso de números con etiquetas y cuadros de texto;
- los fundamentos del uso de cadenas de caracteres.

### ● Introducción

---

En casi todos los programas se utilizan números de un tipo u otro; por ejemplo, para dibujar imágenes mediante el uso de coordenadas en la pantalla, para controlar trayectorias de vuelos espaciales, o para calcular sueldos y deducciones fiscales. En este capítulo veremos los dos tipos básicos de números:

- los números sin decimales, conocidos como enteros en matemáticas, y como el tipo `int` en C#;
- los números con “punto decimal”, conocidos como “reales” en matemáticas, y como el tipo `double` en C#. El término general para los números con punto decimal en computación es números de *punto flotante*.

En el capítulo previo utilizamos valores para producir gráficos en pantalla, pero para realizar programas más sofisticados necesitamos introducir el concepto de una variable: un tipo de caja de almacenamiento que se utiliza para recordar valores, de forma que éstos puedan utilizarse o modificarse más adelante en el programa.

Algunos de los casos en los que se utilizan números `int` son para representar o calcular:

- el número de estudiantes que hay en una clase;
- el número de píxeles que conforman una pantalla;
- el número de copias de un libro que se han vendido hasta cierto momento;

En cuanto a las situaciones que exigen el uso de números `double` podemos señalar:

- mi altura en metros;
- la masa de un átomo en gramos;
- el promedio de los enteros 3 y 4.

Sin embargo, algunas veces el tipo de número a utilizar no es obvio; por ejemplo, si queremos tener una variable para almacenar la calificación de un examen, ¿debemos emplear un número `double` o `int`? No podemos determinar la respuesta con base en lo que sabemos; debemos contar con más detalles. Por ejemplo, podemos preguntar a la persona que se encarga de calificar si redondea al número entero más cercano, o si utiliza números decimales. En consecuencia, la elección entre `int` y `double` se determina a partir de cada problema en particular.

### ● La naturaleza de `int`

Cuando utilizamos un número `int` en C#, puede tratarse de un número entero en el rango de  $-2,147,483,648$  a  $+2,147,483,647$ , o aproximadamente de  $-2,000,000,000$  a  $+2,000,000,000$ .

Todos los cálculos con números `int` son precisos, en cuanto a que toda la información en el número se preserva sin errores.

### ● La naturaleza de `double`

Cuando utilizamos un número `double` en C# su valor puede estar entre  $-1.79 \times 10^{308}$  y  $+1.79 \times 10^{308}$ . En términos no tan matemáticos, el mayor valor es 179 seguido de 306 ceros; ¡sin duda un valor extremadamente grande! Los números se guardan con una precisión aproximada de 15 dígitos.

El principal detalle respecto de las cantidades `double` estriba en que, en casi todos los casos, éstas se guardan en forma aproximada. Para comprender mejor esta característica, realice la siguiente operación en una calculadora:

7 / 3

Si utilizamos siete dígitos (por ejemplo) la respuesta es 2.333333, pero sabemos que una respuesta más exacta sería:

2.3333333333333333

Y aun así, ¡ésa no es la respuesta exacta!

En resumen, como las cantidades `double` se almacenan utilizando un número limitado de dígitos, pueden acumularse pequeños errores en el extremo menos significativo. Para muchos cálculos (por ejemplo, calificaciones de exámenes) esto no es importante, pero para aquellos relacionados con dígitos, el diseño de una nave espacial, cualquier diferencia podría ser relevante. Sin embargo, el rango de precisión de los números `double` es tan amplio que es posible emplearlos sin problemas en los cálculos de todos los días.

Para escribir valores `double` muy grandes (o muy pequeños) se requieren grandes secuencias de ceros. Para simplificar esto podemos usar la notación “científica” o “exponencial”, con `e` o `E`, como en el siguiente ejemplo:

```
double valorGrande = 12.3E+23;
```

lo cual representa 12.3 multiplicado por  $10^{+23}$ . Esta característica se utiliza principalmente en programas matemáticos o científicos.

## ● Declaración de variables

Una vez elegido el tipo de nuestras variables, necesitamos darles un nombre. Podemos imaginarlas como cajas de almacenamiento con un nombre en su exterior y un número (valor) en su interior. El valor puede cambiar a medida que el programa realiza su secuencia de operaciones, pero el nombre es fijo. El programador tiene la libertad de elegir los nombres, y recomendamos escoger aquellos que sean significativos y no crípticos. A pesar de esa libertad, al igual que en casi todos los lenguajes de programación, en C# hay ciertas reglas que debemos seguir. Por ejemplo, los nombres:

- deben empezar con una letra (de la **A** a la **Z** o de la **a** a la **z**);
- pueden contener cualquier cantidad de letras o dígitos (un dígito es cualquier número del 0 al 9);
- pueden contener el guión bajo ‘\_’;
- pueden tener hasta 255 caracteres de longitud.

Tenga en cuenta que C# es sensible al uso de mayúsculas y minúsculas. Por ejemplo, si usted declara una variable llamada `ancho`, no podrá referirse nunca a ella como `Ancho` o `ANCHO`, ya que el uso de las mayúsculas y minúsculas es distinto en cada caso.

Éstas son las reglas de C#, y debemos obedecerlas. Pero también hay un estilo de C#, una forma de usar las reglas que se implementa cuando el nombre de una variable consta de varias palabras: las reglas no permiten separar los nombres con espacios, así que en lugar de utilizar nombres cortos o guiones bajos, el estilo aceptado es poner en mayúscula la primera letra de cada palabra.

Hay otro lineamiento de estilo para decidir si se debe usar mayúscula en la primera letra de un nombre o no. En este capítulo trabajaremos con variables que sólo se utilizan dentro de un método (en vez de que varios métodos las compartan). Las variables de este tipo se conocen como *locales* y sólo se pueden emplear entre los caracteres { y } en donde se declaren. Volviendo a las convenciones de estilo, la metodología de C# dicta *no* poner en mayúscula la primera letra de las variables locales. Más adelante veremos que otros tipos de nombres, como los de métodos y clases, empiezan por convención con letra mayúscula.

Por lo tanto, en vez de:

```
Alturadecaja
h
hob
altura_de_caja
```

usaremos:

```
alturadeCaja
```

He aquí algunos nombres permitidos:

```
cantidad
x
pago2003
```

y éstos son algunos nombres no permitidos (ilegales):

```
2001pago
_area
mi edad
```

También existen algunos nombres reservados para utilización exclusiva de C#, de manera que el programador no los puede reutilizar. Se denominan *palabras clave* o *palabras reservadas*, y ya hemos visto varias de ellas:

```
private
int
new
```

En el apéndice B se incluye una lista completa.

#### PRÁCTICA DE AUTOEVALUACIÓN

**4.1** ¿Cuáles de los siguientes nombres de variables locales están permitidos en C#, y cuáles están escritos en el estilo correcto?

```
volumen
AREA
Longitud
3lados
lado1
longitud
Misalario
su salario
tamanoPantalla
```

El código siguiente corresponde a un programa de ejemplo llamado Área de rectángulo, mismo que analizaremos con detalle a continuación. Supongamos que las medidas de los lados del rectángulo que nos interesa están representadas en números enteros (`int`). Sólo hay un control en el formulario: un botón con el texto “Calcular” en su propiedad `Text`. Todo el código que agregaremos estará dentro del método `button1_Click`.

```
private void button1_Click(object sender, EventArgs e)
{
    int área;
    int longitud;
    int ancho;
    longitud = 20;
    ancho = 10;
    área = longitud * ancho;
    MessageBox.Show("El área es: " + Convert.ToString(área));
}
```

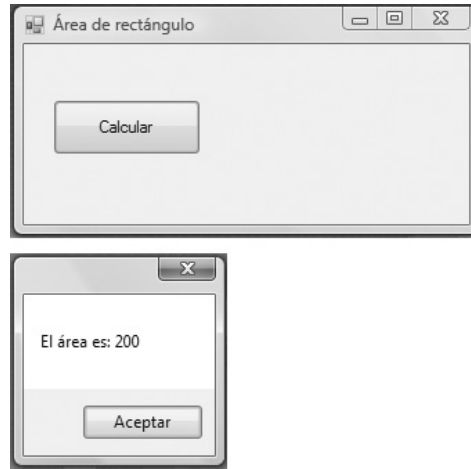


Figura 4.1 El programa Área de rectángulo.

La Figura 4.1 muestra lo que veremos en pantalla al ejecutar el código.

En el programa utilizamos tres variables `int`, que en un momento dado guardarán los datos de nuestro rectángulo. Recuerde que podemos elegir cualquier nombre para nuestras variables; sin embargo, optamos por utilizar nombres claros en vez de nombres cómicos o de una sola letra, que no resultan lo suficientemente claros.

Una vez elegidos los nombres debemos declararlos en el sistema de C#. Aunque esto parece tedioso al principio, el propósito de introducirlos radica en permitir que el compilador detecte errores al escribirlos en el código del programa. He aquí las declaraciones:

```
int área;  
int longitud;  
int ancho;
```

Al declarar variables anteponeamos al nombre que elegimos el tipo que necesitamos (en las tres variables anteriores utilizamos el tipo `int`, de manera que cada variable contendrán un número entero).

También podríamos utilizar como alternativa una sola línea de código, como ésta:

```
int longitud, ancho, área;
```

usando comas para separar cada nombre. Usted puede emplear el estilo de su preferencia; no obstante, le recomendamos usar el primero, ya que nos permite insertar comentarios en cada nombre, en caso de ser necesario. Si usted opta por el segundo estilo, úselo para agrupar nombres relacionados. Por ejemplo, emplee:

```
int alturaImagen, anchoImagen;  
int miEdad;
```

en vez de:

```
int alturaImagen, anchoImagen, miEdad;
```

En casi todos los programas utilizaremos varios tipos, y en C# podemos mezclar las declaraciones, como en el siguiente ejemplo:

```
double alturaPersona;  
int calificaciónExamen;  
double salario;
```

Además es posible establecer el valor inicial de la variable al tiempo que la declaramos, como en estos ejemplos:

```
double alturaPersona = 1.68;  
int a = 3, b = 4;  
int calificaciónExamen = 65;  
int mejorCalificación = calificaciónExamen + 10;
```

Éste es un buen estilo, pero sólo debe usarlo cuando realmente conozca el valor inicial. Si no suministra un valor inicial, C# considera la variable como *no asignada*, y un error de compilación le informará sobre ello si trata de usar su valor en el programa.

## ● La instrucción de asignación

Una vez declaradas nuestras variables podemos colocar nuevos valores en ellas mediante la “instrucción de asignación”, como en el siguiente ejemplo:

```
longitud = 20;
```

El proceso puede visualizarse como se ilustra en la Figura 4.2. Decimos que “el valor 20 se ha asignado a la variable `longitud`”, o que “`longitud` toma el valor de 20”.

Nota:

- El flujo de los datos va de la derecha del signo = hacia la izquierda.
- Cualquier valor que haya tenido `longitud` antes será “sustituido” por 20. Las variables sólo tienen un valor: el actual. Para darle una idea de la velocidad de estas operaciones, considere que una asignación tarda menos de una millonésima de segundo en realizarse.

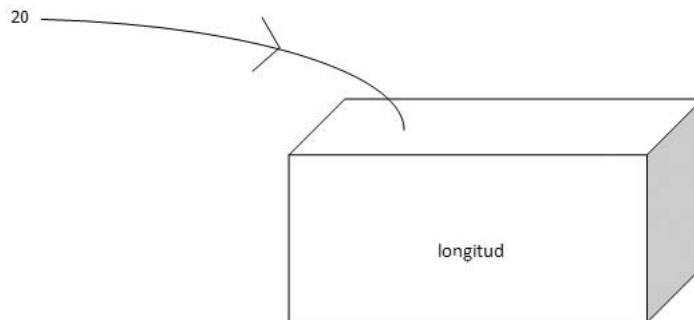


Figura 4.2 Asignación de un valor a una variable.



**PRÁCTICA DE AUTOEVALUACIÓN**

**4.2** Explique el problema que contiene este fragmento de código:

```
int a, b;
a = b;
b = 1;
```

## ● Cálculos y operadores

Veamos de nuevo nuestro programa del rectángulo, en el que se incluye la siguiente instrucción:

```
área = longitud * ancho;
```

La forma general de la instrucción de asignación es:

```
variable = expresión;
```

Una expresión puede tomar varias formas, como un solo número o como un cálculo. En nuestro ejemplo específico la secuencia de eventos es:

1. El carácter `*` hace que se multipliquen los valores almacenados en `longitud` y `ancho`, obteniéndose como resultado el valor 200.
2. El símbolo igual `=` hace que el número 200 se asigne a (se almacene en) `área`.

El carácter `*` es uno de varios “operadores” (se les llama así debido a que operan sobre los valores) y, al igual que en matemáticas, hay reglas para su uso.

Es importante comprender el flujo de los datos, ya que esto nos permite entender el significado de código como el siguiente:

```
int n = 10;
n = n + 1;
```

Lo que ocurre aquí es que la expresión que está al lado derecho del signo `=` se calcula utilizando el valor actual de `n`, con lo cual se obtiene 11. Después este valor se almacena en `n`, sustituyendo el valor anterior, que era 10. Hace algunos años se analizó una gran cantidad de programas, y se descubrió que las instrucciones de la forma:

```
algo = algo + 1;
```

estaban entre las más comunes. De hecho, C# cuenta con una versión abreviada de esta instrucción, llamada instrucción de *incremento*. Los operadores `++` y `--` realizan el incremento y el decremento (o resta de una unidad). Su uso más frecuente es en los ciclos (capítulo 8). He aquí una forma de utilizar el operador `++`:

```
n = 3;
n++;    // ahora n vale 4
```

Respecto del signo `=`, lo importante es saber que no significa “es igual a” en el sentido algebraico. Lo más correcto sería imaginar que significa “toma el valor de” o “recibe”.

## ● Los operadores aritméticos

En esta sección le presentaremos un conjunto básico de operadores: los aritméticos, similares a los botones de cualquier calculadora.

| Operador | Significado    |
|----------|----------------|
| *        | multiplicación |
| /        | división       |
| %        | módulo         |
| +        | suma           |
| -        | resta          |

Observe que dividimos los operadores en grupos para indicar su “precedencia”, es decir, el orden en el que se realizan sus operaciones. Por lo tanto, la multiplicación, división y módulo (\*, / y %) se llevan a cabo antes que la suma y la resta (+ y -). También podemos usar paréntesis para agrupar los cálculos y forzarlos a llevarse a cabo en un orden específico. Si un cálculo incluye operadores de la misma precedencia, las operaciones se realizarán de izquierda a derecha. He aquí algunos ejemplos:

```
int i;
int n = 3;
double d;
i = n + 3;           // se convierte en 6
i = n * 4;           // se convierte en 12
i = 7 + 2 * 4;       // se convierte en 15
n = n * (n + 2) * 4; // se convierte en 60
d = 3.5 / 2;         // se convierte en 1.75
n = 7 / 4;           // se convierte en 1
```

Recuerde que las instrucciones forman una secuencia, la cual se ejecuta de arriba hacia abajo en la página. Siempre que se utilicen paréntesis, los elementos que éstos contengan se calcularán primero. La multiplicación y la división se realizan antes de la suma y la resta. Por lo tanto:

```
3 + 2 * 4
```

se lleva a cabo como si se hubiera escrito así:

```
3 + (2 * 4)
```

Más adelante explicaremos los detalles sobre los operadores / y %.

Observe que, por cuestión de estilo, escribimos un espacio antes y después de un operador. Esto no es esencial, puesto que el programa se ejecutará de todas formas si se omiten los espacios. Sólo los utilizamos para que el programa sea más legible para el programador.

## PRÁCTICA DE AUTOEVALUACIÓN

**4.3** ¿Cuáles son los valores finales de las variables en el siguiente fragmento de código?

```
int a, b, c, d;
d = -8;
a = 1 * 2 + 3;
b = 1 + 2 * 3;
c = (1 + 2) * 3;
c = a + b;
d = -d;
```

Ahora conocemos las reglas. Pero aún hay obstáculos para el principiante. Veamos a continuación algunas fórmulas matemáticas y su conversión a C#. Supongamos que todas las variables están declaradas como tipos `double`, y que su valor inicial ha sido establecido.

| Versión matemática            | Versión de C#                                 |
|-------------------------------|-----------------------------------------------|
| 1 $y = mx + c$                | <code>y = m * x + c;</code>                   |
| 2 $x = (a - b)(a + b)$        | <code>x = (a - b) * (a + b);</code>           |
| 3 $y = 3[(a - b)(a + b)] - x$ | <code>y = 3 * ((a - b) * (a + b)) - x;</code> |
| 4 $y = 1 - \frac{2a}{3b}$     | <code>y = 1 - (2 * a) / (3 * b);</code>       |

En el ejemplo 1 insertamos el símbolo de multiplicación. En C# `mx` se consideraría un nombre de variable.

En el ejemplo 2 necesitamos un signo de multiplicación explícito entre los paréntesis.

En el ejemplo 3 sustituimos los corchetes matemáticos por paréntesis.

En el ejemplo 4 podríamos haber cometido el error de usar esta versión incorrecta:

```
y = 1 - 2 * a / 3 * b;
```

Recuerde la regla según la cual los cálculos se realizan de izquierda a derecha cuando los operadores tienen igual precedencia. El problema tiene que ver con los operadores `*` y `/`. El orden de evaluación es como si hubiéramos utilizado:

```
y = 1 - (2 * a / 3) * b;
```

es decir, la `b` ahora está multiplicando en vez de dividir. La forma más simple de manejar los cálculos potencialmente confusos consiste en utilizar paréntesis adicionales; hacerlo no implica penalización alguna en términos de tamaño o velocidad del programa.

El uso de los operadores `+`, `-` y `*` es razonablemente intuitivo, pero la división es un poco más engañosa, ya que exige diferenciar entre los tipos `int` y `double`. En este sentido, lo importante es tomar en cuenta que:

- Cuando el operador `/` trabaja con dos números `double` o con una mezcla de `double` e `int` se produce un resultado `double`. Para fines de cálculo, cualquier valor `int` se considera como `double`. Así es como funciona la división en una calculadora de bolsillo.

- Cuando / trabaja con dos enteros se produce un resultado entero. El resultado se trunca, lo cual significa que se borran los dígitos que pudiera haber después del “punto decimal”. Ésta *no* es la forma en que funcionan las calculadoras.

He aquí algunos ejemplos:

```
// división con valores double
double d;
d = 7.61 / 2.1;    // se convierte en 3.7
d = 10.6 / 2;      // se convierte en 5.3
```

En el primer caso la división se lleva a cabo de la manera esperada.

En el segundo el número 2 se trata como 2.0 (es decir, un `double`) y la división se realiza.

Sin embargo, la división con enteros es distinta:

```
//división con enteros
int i;
i = 10 / 5;        // se convierte en 2
i = 13 / 5;        // se convierte en 2
i = 33 / 44;       // se convierte en 0
```

En el primer caso se espera una división con enteros; la respuesta exacta que se produce es 2.

En el segundo caso el resultado también es 2, debido a que se trunca el verdadero resultado.

En el tercer caso se trunca la respuesta “correcta” de 0.75, con lo cual obtenemos 0.

#### PRÁCTICAS DE AUTOEVALUACIÓN

**4.4** Mi salario es de \$20,000 y estoy de acuerdo en darle a usted la mitad del mismo utilizando el siguiente cálculo:

```
int mitad = 20000 * (1 / 2);
```

¿Cuánto recibirá usted?

**4.5** Indique los valores con los que terminan a, b, c y d después de realizar los siguientes cálculos:

```
int a, b, c, d;
a = 7 / 3;
b = a * 4;
c = (a + 1) / 2;
d = c / 3;
```

#### ● El operador %

Por último veremos el operador % (módulo). A menudo se utiliza junto con la división de enteros, ya que provee la parte del residuo. Su nombre proviene del término “módulo” que se utiliza en una rama de las matemáticas conocida como aritmética modular.

Anteriormente dijimos que los valores `double` se almacenan de manera aproximada, a diferencia de los enteros, que lo hacen de forma exacta. Entonces ¿cómo puede ser que **33/44** genere un resul-

tado entero de 0? ¿Acaso perder 0.75 significa que el cálculo no es preciso? La respuesta es que los enteros *sí* operan con exactitud, pero el resultado exacto está compuesto de dos partes: el cociente (es decir, el resultado principal) y el residuo. Por lo tanto, si dividimos 4 entre 3 obtenemos como resultado 1, con un residuo de 1. Esto es más exacto que 1.3333333, etc.

En consecuencia, el operador % nos da el residuo como si se hubiera llevado a cabo una división. He aquí algunos ejemplos:

```
int i;
double d;
i = 12 % 4;           // se convierte en 0
i = 13 % 4;           // se convierte en 1
i = 15 % 4;           // se convierte en 3
d = 14.9 % 3.9;       // se convierte en 3.2 (se divide 3.2 veces)
```

Hasta ahora el uso más frecuente de % es con números `int`, pero cabe mencionar que también funciona con números `double`. Veamos un problema que involucra un resultado con residuo: convertir un número entero de centavos en dos cantidades: la cantidad de dólares y el número de centavos restantes. La solución es:

```
int centavos = 234;
int dólares, centavosRestantes;
dólares = centavos / 100;           // se convierte en 2
centavosRestantes = centavos % 100; // se convierte en 34
```

#### PRÁCTICA DE AUTOEVALUACIÓN

**4.6** Complete el siguiente fragmento de código. Agregue instrucciones de asignación para dividir `totalSegundos` en dos variables: `minutos` y `segundos`.

```
int totalSegundos = 307;
```

## ● Unión de cadenas con el operador +

Hasta ahora hemos visto el uso de variables numéricas, pero también es muy importante el procesamiento de datos de texto. C# cuenta con el tipo de datos `string`; las variables `string` pueden guardar cualquier carácter. La longitud máxima de una cadena es de aproximadamente dos mil millones de caracteres; cantidad superior al tamaño de la RAM de las computadoras actuales.

El siguiente es un ejemplo del uso de cadenas:

```
string primerNombre = "Mike ";
string apellidoPaterno, nombreCompleto;
string saludo;
apellidoPaterno = "Parr";
nombreCompleto = primerNombre + apellidoPaterno;
saludo = "Saludos de " + nombreCompleto; //se convierte en "Saludos de Mike Parr"
```

En el ejemplo anterior declaramos algunas variables `string` y les asignamos valores iniciales mediante el uso de comillas dobles. Después utilizamos la asignación, en la cual el valor de la cadena a la derecha del signo `=` se almacena en la variable utilizada a la izquierda del mismo, de manera similar a la asignación numérica (si intentamos utilizar una variable que no haya recibido un valor, C# nos informará que la variable no está asignada y el programa no se ejecutará).

Las siguientes líneas ilustran el uso del operador `+`, que (de igual manera que al sumar números) opera sobre las cadenas y las une extremo con extremo. A esto se le conoce como “concatenación”. Después de la instrucción:

```
nombreCompleto = primerNombre + apellidoPaterno;
```

el valor de `nombreCompleto` es `Mike Parr`.

Además hay un amplio rango de métodos de cadenas que proporcionan operaciones tales como búsqueda y modificación de cadenas. Hablaremos sobre estos métodos en el capítulo 16.

Previamente se mencionó que el operador `/` considera los elementos que divide como números `double` si uno de ellos es `double`. El operador `+` trabaja de manera similar con las cadenas. Por ejemplo:

```
int i = 7;
string nombre = "a. avenida";
string s = i + nombre;
```

En este caso el operador `+` detecta que `nombre` es una variable `string` y convierte `i` en una cadena antes de unir ambas variables. Éste es un método abreviado conveniente para evitar la conversión explícita que veremos más adelante, pero puede resultar engañoso. Considere el siguiente código:

```
int i = 2, j = 3;
string s, nota = "La respuesta es: ";
s = nota + i + j;
```

¿Cuál es el valor de `s`? Las dos posibilidades son:

- La respuesta es: 23, en donde ambos operadores `+` trabajan sobre cadenas.
- La respuesta es: 5, en donde el segundo `+` suma números.

De hecho lo que ocurre es el primer caso. C# trabaja de izquierda a derecha. El primer `+` produce la cadena “La respuesta es: 2”; después, el segundo `+` agrega el 3 a la derecha. No obstante, si colocamos:

```
s = nota + (i + j);
```

primero se calcula la operación `2 + 3`, obteniéndose 5. Por último se lleva a cabo la unión de las cadenas.

## PRÁCTICA DE AUTOEVALUACIÓN

### 4.7 ¿Qué despliegan en pantalla los siguientes cuadros de mensaje?

```
MessageBox.Show("5"+"5"+5+5);
MessageBox.Show("5"+"5"+(5+5));
```

## ● Conversiones entre cadenas y números

Uno de los usos más importantes del tipo de datos `string` son las operaciones de entrada y salida, en donde procesamos los datos que introduce el usuario y desplegamos los resultados en pantalla. Muchos de los controles de la GUI de C# trabajan con cadenas de caracteres en vez de hacerlo con números, por lo cual es preciso que aprendamos a realizar conversiones entre números y cadenas. La clase `Convert` proporciona varios métodos convenientes para ese propósito.

Para convertir una variable o cálculo (una expresión en general) podemos utilizar el método `ToString`. He aquí algunos ejemplos:

```
string s1, s2;
int num = 44;
double d=1.234;
s1 = Convert.ToString(num);    // s1 es "44"
s2 = Convert.ToString(d);     // s2 es "1.234"
```

Por lo general el nombre del método va precedido por el de un objeto con el que debe trabajar, pero aquí suministramos el objeto como un argumento entre paréntesis. Los métodos que funcionan de esta forma se denominan estáticos (`static`); cada vez que los utilicemos deberemos identificar la clase a la que pertenecen. Éste es el motivo por el que colocamos `Convert` antes de `ToString`. En el capítulo 10 analizaremos más a fondo los métodos `static`.

En el ejemplo anterior el método `ToString` nos regresa una cadena que podemos almacenar en una variable, o utilizarla de alguna otra forma.

En el programa para calcular el área de un rectángulo utilizamos el operador `+` y el método `ToString` con un cuadro de mensaje desplegable. En vez de mostrar sólo el número, lo unimos a un mensaje:

```
MessageBox.Show("El área del rectángulo es: " + Convert.ToString(área));
```

El siguiente código no compila, ya que el método `Show` espera un valor `string` como parámetro:

```
MessageBox.Show(área);    //NO - ¡no compilará!
```

Debemos utilizar:

```
MessageBox.Show(Convert.ToString(área));
```

Para complementar el método `ToString` tenemos los métodos `ToInt32` y `ToDouble`, los cuales convierten las cadenas de caracteres en números. Observe que no hay un método `ToInt`. La clase `Convert` está disponible para cualquier lenguaje que utilice el marco de trabajo (framework) .NET, y el nombre de clase a nivel de marco de trabajo para los elementos `int` en C# es `Int32` (enteros de 32 bits). He aquí algunos ejemplos:

```
double d;
int i;
string s1 = "12.3";
string s2 = "567";
d = Convert.ToDouble(s1);
i = Convert.ToInt32(s2);
```

**PRÁCTICA DE AUTOEVALUACIÓN**

**4.8** ¿Cuáles son los valores finales de `m`, `n` y `s` en el código siguiente?

```
int m, n;
string s;
string v = "3";
m = Convert.ToInt32(v + v + "4");
n = Convert.ToInt32(v + v) + 4;
s = Convert.ToString(Convert.ToInt32(v)
    + Convert.ToInt32(v)) + "4";
```

Ahora que sabemos realizar conversiones de cadenas, podemos empezar a utilizar varios controles nuevos.

### ● Cuadros de texto y etiquetas

En los programas en que hemos venido trabajando utilizamos instrucciones de asignación con el propósito de establecer valores iniciales para los cálculos; pero, en la práctica no conoceremos esos valores al escribir el programa, ya que el usuario los introducirá a medida que éste se vaya ejecutando. En esta sección veremos el control `TextBox`, el cual permite que un usuario introduzca datos, y el control `Label` que se utiliza para desplegar información (por ejemplo, los resultados de un cálculo, o instrucciones para el usuario) en un formulario.

Como sabemos, para usar un cuadro de texto todo lo que tenemos que hacer es seleccionarlo en el cuadro de herramientas y colocarlo en un formulario. Estos controles tienen muchas propiedades, pero la principal es `Text`, que nos proporciona la cadena escrita por el usuario. Para acceder a esta propiedad utilizamos la ya conocida notación de “punto”, como en el siguiente ejemplo:

```
string s;
s = textBox1.Text;
```

Es bastante común que el programador elimine el contenido de la propiedad `Text` del control en tiempo de diseño (mediante la ventana de propiedades), para que el usuario pueda escribir en un área en blanco.

Al igual que en el caso de los cuadros de texto, la principal propiedad del control `Label` (disponible también en el cuadro de herramientas) es `Text`, pues nos permite establecer la cadena que la etiqueta mostrará en pantalla. Podemos acceder a esta propiedad de la siguiente manera:

```
string s = "Alto";
label1.Text = s;
```

Algunas etiquetas se utilizan para mostrar mensajes de ayuda al usuario; por lo general establecemos su propiedad `Text` en tiempo de diseño mediante la ventana de propiedades. No es necesario que el texto que contienen cambie durante la ejecución del programa. Por otro lado, en el caso de las etiquetas que despliegan resultados hay que establecer su propiedad `Text` en tiempo de ejecución, como se muestra en el ejemplo anterior. El usuario puede sobrescribir los cuadros de texto, pero las etiquetas están protegidas.

En general, las clases tienen métodos y propiedades. Los métodos hacen que los objetos realicen acciones, mientras que las propiedades nos permiten acceder al estado actual de un objeto. He aquí



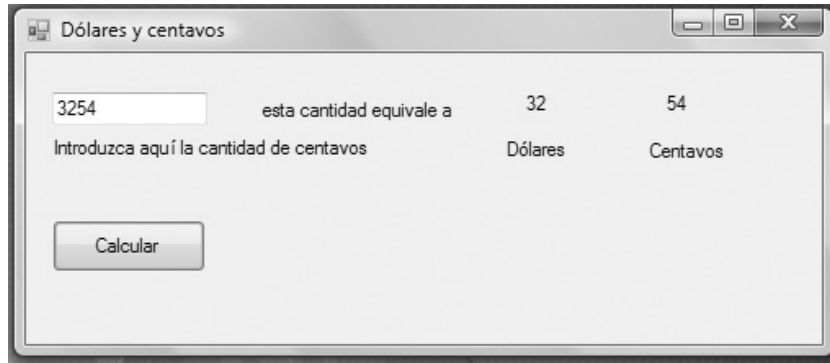


Figura 4.3 El programa Dólares y centavos.

un programa de ejemplo (Dólares y centavos), en el que una cantidad en centavos se convierte a dólares y centavos. Anteriormente en este capítulo vimos cómo usar los operadores `/` y `%`. En la Figura 4.3 se muestra este programa en ejecución; en él se utiliza un cuadro de texto y varias etiquetas.

```
private void button1_Click(object sender, EventArgs e)
{
    int centavos;
    centavos = Convert.ToInt32(textBox1.Text);
    dólaresEtiqueta.Text = Convert.ToString(centavos / 100);
    centavosEtiqueta.Text = Convert.ToString(centavos % 100);
}
```

Los principales controles que utilizamos en este programa son:

- un botón para iniciar la conversión;
- un cuadro de texto en donde el usuario introduce una cantidad en centavos;
- dos etiquetas: para mostrar el número de dólares y el número de centavos.

Además hay tres etiquetas debajo del cuadro de texto y las dos etiquetas que muestran el resultado para ayudar al usuario a entender el formulario. Los valores de texto de las etiquetas son:

```
Introduzca aquí la cantidad de centavos
Dólares
Centavos
```

Como vimos en el capítulo 2, es recomendable cambiar el nombre de los controles cuando hay más de una instancia del mismo tipo de control en un formulario. En este programa:

- hay un botón y un cuadro de texto, por lo que podemos dejar a estos controles el nombre que C# les asignó;
- hay dos etiquetas que muestran resultados. Como tener dos controles Etiqueta podría causar confusión, les damos un nombre específico a cada uno de ellos;
- al resto de las etiquetas se les asigna su propiedad de texto en tiempo de diseño, y el programa nunca las manipulará. Podemos dejar a estas etiquetas los nombres que C# les asignó.

He aquí un resumen de las principales propiedades de los controles.

| Control          | Propiedad | Valor    |
|------------------|-----------|----------|
| button1          | Text      | Calcular |
| textBox1         | Text      | (vacía)  |
| dólaresEtiqueta  | Text      | (vacía)  |
| centavosEtiqueta | Text      | (vacía)  |

Recuerde que es conveniente cambiar el nombre de los controles tan pronto como los coloque en el formulario, antes de hacer doble clic para crear el código de cualquier evento.

Cuando el programa se ejecuta el usuario introduce un número en el cuadro de texto. Al hacer clic en el botón se lleva a cabo el cálculo y los resultados se colocan en las dos etiquetas. Sin embargo, hay que realizar conversiones de cadenas a números y viceversa. He aquí un extracto:

```
centavos = Convert.ToInt32(textBox1.Text);
dólaresEtiqueta.Text = Convert.ToString(centavos / 100);
```

El programa ilustra el uso de un cuadro de texto y de etiquetas para mostrar resultados que pueden cambiar, junto con mensajes que no se modifican.

#### PRÁCTICA DE AUTOEVALUACIÓN

**4.9** Sabemos que pueden utilizarse tanto cuadros de mensaje como etiquetas para mostrar resultados. ¿Cuál es la principal diferencia entre ambas opciones?

### ● Conversiones entre números

Habrà ocasiones en que necesitaremos convertir valores numéricos de un tipo a otro. Los casos más comunes son la conversión de un `int` a un `double` y viceversa.

Veamos un ejemplo: tenemos nueve manzanas y queremos repartirlas de manera equitativa entre cuatro personas. Sin duda los valores 9 y 4 son enteros, pero la respuesta es un valor `double` (es decir, incluye decimales). Para resolver este problema debemos conocer algunos fundamentos sobre la conversión numérica.

Veamos primero algunos ejemplos de conversiones:

```
int i = 33;
double d = 3.9;
double d1;
d1 = i;           // se convierte en 33.0
// o, de manera explícita:
d1 = (double)i;   // se convierte en 33.0
i = (int)d;       // se convierte en 3
```

Los puntos a tomar en cuenta son:

- Asignar un `int` a un `double` no requiere programación adicional. Es un proceso seguro, ya que no se puede perder información; no hay posiciones decimales por los cuales preocuparse.
- Al asignar un `double` a un `int` pueden perderse los dígitos que suceden al punto decimal, ya que no caben en el entero. Debido a esta pérdida potencial de información, C# requiere que especifiquemos esta conversión de manera explícita. Podríamos utilizar la clase `Convert` para solucionar la situación, pero mejor usaremos otro método, conocido como conversión de tipos o *casting* (emplearemos también esta característica cuando veamos las herramientas más avanzadas de la programación orientada a objetos).
- Para convertir un `double` a la forma de un `int` debemos anteponer la palabra (`int`). En ese caso el valor se truncará al eliminar los dígitos que suceden al punto decimal.
- Cabe mencionar que podríamos usar una conversión explícita de tipos al convertir un `int` en un `double`, pero esto no es necesario.

Volviendo a nuestro ejemplo de las manzanas, podemos obtener una respuesta `double` si utilizamos las siguientes líneas de código:

```
int manzanas = 9; //u obtener el valor a partir de un cuadro de texto
int personas = 4; //u obtener el valor a partir de un cuadro de texto
MessageBox.Show("Cada persona recibe: " + Convert.ToString(
    (double)manzanas / (double)personas));
```

Observe que `(double)(manzanas / personas)` produciría la respuesta incorrecta, ya que se realizaría una división entre enteros.

#### PRÁCTICA DE AUTOEVALUACIÓN

**4.10** ¿Cuáles son los valores de `a`, `b`, `c`, `i`, `j` y `k` después de ejecutar el siguiente código?

```
int i, j, k;
double a, b, c;
int n = 3;
double y = 2.7;
i = (int)y;
j = (int)(y + 0.6);
k = (int)((double)n + 0.2);
a = n;
b = (int)n;
c = (int)y;
```

## ● Función de las expresiones

Aunque hemos recalcado que las expresiones (cálculos) pueden formar el lado derecho de las instrucciones de asignación, también es posible ubicarlas en otros lugares. De hecho, podemos colocar una expresión `int` en cualquier parte en donde se pueda poner un valor `int` individual. Considere el método `DrawLine` que vimos en ejemplos anteriores, el cual requiere cuatro enteros para especificar el inicio y el final de la línea a dibujar. Podríamos (si fuera conveniente) sustituir los números con variables o con expresiones:

```
int x = 100;
int y = 200;
papel.DrawLine(lápiz, 100, 100, 110, 110);
papel.DrawLine(lápiz, x, y, x + 50, y + 50);
papel.DrawLine(lápiz, x * 2, y - 8, x * 30 - 1, y * 3 + 6);
```

Las expresiones se calculan y los valores resultantes se pasan a `DrawLine` para que los utilice.

## Fundamentos de programación

- Las variables tienen un nombre, y el programador puede elegir el que desee asignarles.
- Las variables tienen un tipo, y el programador puede elegir cuál de ellos utilizará.
- Las variables contienen un valor.
- El valor de una variable puede modificarse mediante una instrucción de asignación.

## Errores comunes de programación

- Tenga cuidado al escribir los nombres de las variables. Por ejemplo, en:

```
int círculo;    // error de escritura
círculo = 20;
```

la variable está mal escrita en la primera línea, ya que se utiliza un ‘l’ (uno) en vez de una ‘L’ minúscula. En este caso el compilador de C# detectará que la variable de la segunda línea no está declarada. Otro error común es utilizar un cero en vez de una ‘O’ mayúscula.

- Es difícil detectar los errores de compilación al principio. Aunque el compilador de C# nos da una señal de la posición en la que cree que se encuentra el error, en realidad éste podría hallarse en una línea anterior.
- Los paréntesis deben estar balanceados, es decir, debe haber el mismo número de ‘(’ que de ‘)’.
- Al utilizar números con la propiedad de texto de las etiquetas y los cuadros de texto, recuerde utilizar las herramientas de conversión de cadenas.
- Al multiplicar elementos debe colocar el carácter `*` entre ellos, mientras que en matemáticas se omite este signo. Al dividir elementos, recuerde que:

- `int / int` nos da una respuesta `int`.
- `double / double` nos da una respuesta `double`.
- `int / double` y `double / int` nos dan una respuesta `double`.

## Secretos de codificación

- Para declarar variables indicamos su clase y su nombre; por ejemplo:

```
int miVariable;
string tuVariable = "¡Saludos a todos!";
```

- Los tipos de variables más útiles son `int`, `double` y `string`.
- Los principales operadores aritméticos son `*`, `/`, `%`, `+` y `-`.
- El operador `+` se utiliza para unir cadenas.
- Los operadores `++` y `--` pueden emplearse para incrementar y decrementar.
- Podemos convertir números a cadenas con el método `Convert.ToString`.
- Podemos convertir cadenas a números con los métodos `Convert.ToInt32` y `Convert.ToDouble`.
- Si colocamos el operador de conversión (`int`) antes de un elemento `double`, éste se convierte en un entero.
- Si colocamos el operador de conversión (`double`) antes de un elemento `int`, éste se convierte en un valor `double`.

## Nuevos elementos del lenguaje

- `int double string`
- los operadores `+` `-` `*` `/` `%`
- `++` y `--` para incremento y decremento
- `=` para asignación
- Conversión de tipos: la clase `Convert`, los operadores de conversión (`double`) e (`int`).

## Nuevas características del IDE

- Los controles `TextBox` y `Label`, con sus propiedades `Text`.
- La posibilidad de cambiar el nombre de los controles.

## Resumen

- Las variables se utilizan para contener (guardar) valores. Mantienen su valor hasta que éste es modificado de manera explícita (por ejemplo, mediante otra instrucción de asignación).
- Los operadores operan sobre valores.
- Las expresiones son cálculos que producen un valor. Pueden utilizar en una variedad de situaciones, incluyendo al lado derecho de una asignación, o como argumentos para invocar un método.

## EJERCICIOS

**4.1** Amplíe el programa del rectángulo que vimos en este capítulo para que calcule el volumen de una caja a partir de sus tres dimensiones.

**4.2** (a) Dado el siguiente valor:

```
double radio = 7.5;
```

utilice instrucciones de asignación para calcular la circunferencia de un círculo, el área de un círculo y el volumen de una esfera con base en el mismo radio. Despliegue los resultados en cuadros de mensaje. Los mensajes deben indicar cuál es el resultado en vez de sólo mostrar un número. Estos cálculos implican el uso de Pi, cuyo valor aproximado es 3.14. Sin embargo, C# nos proporciona este valor con más dígitos de precisión en la clase `Math`; la fórmula siguiente muestra cómo se utiliza:

```
circunferencia = 2 * Math.PI * radio;  
área = Math.PI * radio * radio;  
volumen = (4 * Math.PI / 3) * radio * radio * radio;
```

(b) Modifique la parte (a) de manera que se utilice un cuadro de texto para introducir el radio, además de etiquetas para los resultados. Use etiquetas adicionales para mejorar la presentación de los resultados.

**4.3** Dos estudiantes presentan un examen de C#, y sus resultados se asignan a dos variables:

```
int calificación1 = 44;  
int calificación2 = 51;
```

Escriba un programa que calcule y muestre la calificación promedio como un valor `int`. Verifique su resultado con una calculadora.

**4.4** Dos estudiantes presentan un examen de C#, y sus resultados —el profesor es muy estricto— son valores `double`. Escriba un programa que calcule y muestre la calificación promedio como un valor `double`. Verifique su respuesta con una calculadora.

**4.5** Suponga que un grupo de personas tienen que pagar impuestos de 20% sobre sus ingresos. Obtenga el valor del ingreso de un cuadro de texto. Después calcule y despliegue la cantidad inicial, la cantidad después de las deducciones y la cantidad que se dedujo. Use etiquetas para que los resultados se entiendan.

**4.6** Utilice tipos `int` para escribir un programa que convierta una temperatura en grados Fahrenheit a su equivalente en Celsius (centígrados). La fórmula es:

```
c = (f - 32) * 5 / 9
```

**4.7** Dado un número inicial de segundos:

```
int totalSegundos = 5049;
```

Escriba un programa para convertirlos a horas, minutos y segundos. Prepare un ejemplo con pluma y papel antes de escribir el programa. Use un cuadro de mensaje para desplegar el resultado de la siguiente forma:

```
H:1 M:24 S:9
```

**4.8** Este problema está relacionado con las resistencias eléctricas, las cuales “resisten” el flujo de la corriente eléctrica que pasa a través de ellas. Las mangueras son una analogía: una manguera delgada tiene una alta resistencia, y una gruesa tiene una baja resistencia al agua. Imagine que tenemos dos mangueras, si las conectamos en serie se obtendría una mayor resistencia, y si las conectamos en paralelo se reduciría la resistencia (ya que obtendríamos el equivalente a una manguera más gruesa). Dadas las siguientes declaraciones:

```
double r1 = 4.7;
double r2;
```

calcule y muestre la resistencia en serie con base en:

```
series = r1 + r2
```

y la resistencia en paralelo de acuerdo con:

$$\text{paralelo} = \frac{r1 * r2}{r1 + r2}$$

**4.9** Suponga que necesitamos instalar cierto software en una máquina europea dispensadora de bebidas. He aquí los detalles: todos los elementos cuestan menos de 1 euro (100 centavos de euro) y la denominación más alta que podemos insertar es una moneda de 1 euro. Dado el monto insertado y el costo del artículo su programa debe regresar cambio utilizando el menor número de monedas. Por ejemplo, si tenemos que:

```
int montoDado = 100;
int costoArtículo = 45;
```

el resultado debería ser una serie de cuadros de mensaje (uno para cada moneda) de la siguiente forma:

```
La cantidad de monedas de 50 centavos es 1
La cantidad de monedas de 20 centavos es 0
La cantidad de monedas de 10 centavos es 0
La cantidad de monedas de 5 centavos es 1
La cantidad de monedas de 2 centavos es 0
La cantidad de monedas de 1 centavos es 0
```

Sugerencia: trabaje con centavos y utilice el operador % todas las veces que pueda. Las monedas de euro son: 100, 50, 20, 10, 5, 2, 1.

**SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**

- 4.1** volumen – permitido, estilo correcto  
AREA – permitido, pero es preferible usar area  
Longitud – permitido, pero es preferible usar la 'l' minúscula  
3lados – no está permitido, ya que empieza con un dígito  
lado1 – permitido, estilo correcto  
lonitud – permitido, aun cuando está mal escrita la palabra 'longitud'  
Misalario – permitido, pero es preferible usar misalario  
su salario – no permitido (están prohibidos los espacios en medio de un nombre)  
tamanoPantalla – permitido, estilo correcto
- 4.2** En la línea 2, b no está asignada. Se producirá un error de compilación debido a que estamos tratando de almacenar una variable no asignada en a.
- 4.3** Los valores finales de a, b, c y d son 5, 7, 12 y 8, respectivamente.
- 4.4** Por desgracia usted no recibe nada, ya que primero se calcula  $(1 / 2)$  y se obtiene 0. Es mejor que utilice 0.5.
- 4.5** Los valores finales de a, b, c y d son 2, 8, 1 y 0, respectivamente.
- 4.6**
- ```
int totalSegundos = 307;  
int segundos, minutos;  
minutos = totalSegundos / 60;  
segundos = totalSegundos % 60;
```
- 4.7** Los cuadros de mensaje muestran las cadenas 5555 y 5510, respectivamente.  
En el primer caso procedemos de izquierda a derecha, uniendo cadenas.  
En el segundo se llevan a cabo las operaciones dentro de los paréntesis y se obtiene el entero 10. Después se lleva a cabo la unión de cadenas.
- 4.8** Los valores finales de m, n y s son 334, 37 y 64, respectivamente.
- 4.9** Una etiqueta muestra sus resultados en el formulario y no requiere interacción por parte del usuario. Un cuadro de mensaje aparece y el usuario debe hacer clic en "Aceptar" para cerrarlo; en otras palabras, el cuadro de mensaje obliga al usuario a enterarse de su presencia.
- 4.10** Los valores de las variables `int i, j y k` son 2, 3 y 3, y los valores de las variables `double a, b y c` son 3.0, 3.0 y 2.0, respectivamente.



# Métodos y argumentos

En este capítulo conoceremos cómo:

- escribir métodos;
- utilizar argumentos y parámetros;
- pasar argumentos por valor y por referencia;
- usar **return** en los métodos.

## ● Introducción

Los programas grandes pueden ser complejos, difíciles de comprender y de depurar. La técnica más importante para reducir esta complejidad consiste en dividir el programa en secciones (relativamente) independientes. Esto nos permite enfocarnos en una sección específica sin distraernos con el programa completo. Además, si la sección tiene nombre podemos “llamarla” o “invocarla” (es decir, hacer que sea utilizada por otra instancia) con sólo hacer referencia a ella. Trabajar de esta manera nos permite, en cierto sentido, pensar a un nivel más alto. En C# a dichas secciones se les conoce como métodos.

Veamos un ejemplo: en el capítulo 3 utilizamos una buena cantidad de métodos gráficos predefinidos para dibujar figuras en pantalla, como el método `DrawRectangle`, al cual podemos invocar con cinco argumentos de la siguiente forma:

```
papel.DrawRectangle(miLápiz, 10, 20, 60, 60);
```

Al utilizar argumentos (los elementos entre paréntesis) podemos controlar el tamaño y la posición del rectángulo, y garantizar que `DrawRectangle` será lo suficientemente flexible como para funcionar en diversas circunstancias. Los argumentos modifican sus acciones.

Además, cabe mencionar que podríamos producir un rectángulo mediante el uso de cuatro llamadas a `DrawLine`. Sin embargo, es mucho más sensato agrupar las cuatro instrucciones `DrawLine` en un método conocido como `DrawRectangle`, ya que hacerlo de esta manera permite que el programador aproveche al máximo sus habilidades.

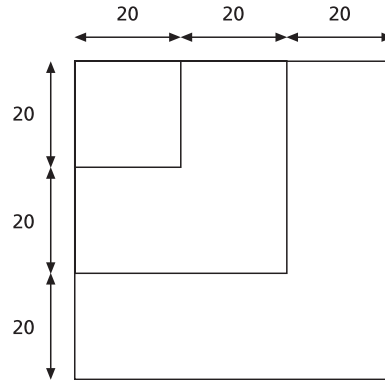


Figura 5.1 El logotipo de la empresa.

## ● Creación de métodos propios

En esta sección hablaremos acerca de cómo crear nuestros propios métodos. Empezaremos con un pequeño ejemplo para simplificar las cosas, y después veremos un caso más práctico.

La Compañía Mundial de Cajas de Cartón tiene un logotipo que consiste en tres cuadrados, uno dentro de otro, como se muestra en la Figura 5.1.

Los responsables de la empresa desean utilizar el logotipo en varias posiciones dentro de un cuadro de imagen, como se muestra en la Figura 5.2. He aquí el código para dibujar dos logotipos idénticos en las posiciones (10, 20) y (100, 100).

```
// Dibuja el logotipo en la esquina superior izquierda
papel.DrawRectangle(miLápiz, 10, 20, 60, 60);
papel.DrawRectangle(miLápiz, 10, 20, 40, 40);
papel.DrawRectangle(miLápiz, 10, 20, 20, 20);

// Dibuja el logotipo en la esquina superior derecha
papel.DrawRectangle(miLápiz, 100, 100, 60, 60);
papel.DrawRectangle(miLápiz, 100, 100, 40, 40);
papel.DrawRectangle(miLápiz, 100, 100, 20, 20);
```

Observe que los cuadrados son de 20, 40 y 60 píxeles, y que su esquina superior izquierda está en el mismo punto. Si analiza el código observará que, en esencia, se repiten las tres instrucciones para dibujar el logotipo, independientemente de la posición de su esquina superior izquierda. A continuación agruparemos esas tres instrucciones para crear un método, de manera que pueda dibujarse un logotipo con una sola instrucción.

## ● Nuestro primer método

El siguiente es el código de un programa completo llamado Método logotipo. Este programa muestra cómo crear y utilizar un método, al cual denominaremos `DibujarLogo`. La convención de estilo de C# nos recomienda empezar los nombres de los métodos con mayúscula.

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    Pen miLápiz = new Pen(Color.Black);
    DibujarLogo(papel, miLápiz, 10, 20);
    DibujarLogo(papel, miLápiz, 100, 100);
}

private void DibujarLogo(Graphics áreaDibujo,
    Pen lápizAUsar,
    int posX,
    int posY)
{
    áreaDibujo.DrawRectangle(lápizAUsar, posX, posY, 60, 60);
    áreaDibujo.DrawRectangle(lápizAUsar, posX, posY, 40, 40);
    áreaDibujo.DrawRectangle(lápizAUsar, posX, posY, 20, 20);
}
```

El programa consta de un cuadro de imagen y un botón. Al hacer clic en el botón se dibujan dos logotipos, como se muestra en la Figura 5.2.

El concepto de los métodos y argumentos es una importante habilidad que los programadores necesitan dominar. En la sección siguiente analizaremos a detalle el programa. Considere el siguiente extracto:

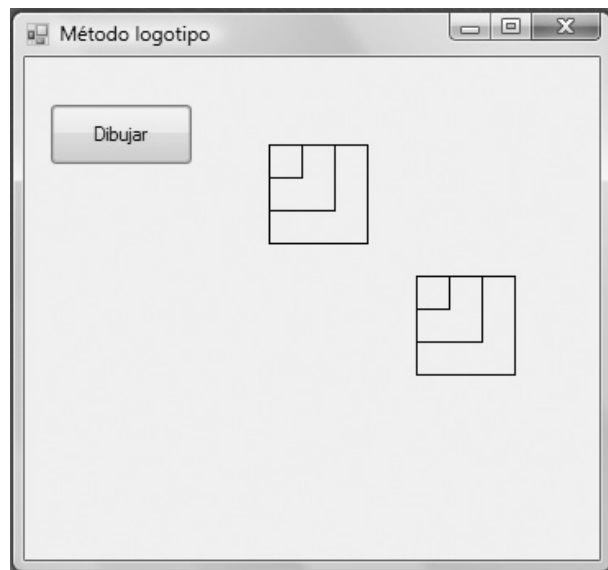


Figura 5.2 El programa Método logotipo.

```
private void DibujarLogo(Graphics áreaDibujo,  
    Pen lápizAUsar,  
    int posX,  
    int posY)
```

Aquí se declara (introduce) el método; a esto se le conoce como encabezado del método. El encabezado declara el nombre del método (que nosotros tenemos la libertad de elegir) y los elementos que deben suministrarse para controlar su operación. C# utiliza los términos *argumentos* y *parámetros* para definir estos elementos; a continuación hablaremos de ellos. Al resto del método se le conoce como *cuerpo*, y va encerrado entre llaves { }; aquí es donde se realiza el trabajo. A menudo el encabezado es una línea extensa, y nosotros podemos optar por dividirlo en puntos adecuados (aunque no en medio de una palabra).

Una importante decisión que debe tomar el programador es el lugar desde donde se puede invocar el método; en este sentido, tenemos dos opciones:

- El método sólo puede ser invocado desde el interior del programa actual; en este caso utilizamos la palabra clave `private`.
- El método puede ser invocado desde otro programa; en este caso utilizamos la palabra clave `public`. Los métodos como `DrawRectangle` son ejemplos de métodos que se han declarado como `public`; son de uso general (para crear métodos `public` o públicos se requiere un conocimiento más profundo de la programación orientada a objetos; hablaremos al respecto con más detalle en el capítulo 10).

Otra decisión que debe tomar el programador es:

- ¿El método realizará una tarea sin necesidad de producir un resultado? En este caso utilizamos la palabra clave `void` después de `private`.
- ¿El método calculará un resultado y lo devolverá a la sección de código que lo invocó? En este caso tenemos que declarar el tipo del resultado, en vez de usar `void`. Más adelante en el capítulo veremos cómo hacerlo.

En el caso del método `DibujarLogo`, su tarea consiste en dibujar líneas en la pantalla, y no en proveer la respuesta de un cálculo. Por ende, utilizamos `void`.

## ● Cómo invocar métodos

---

Para invocar un método privado en C# es preciso indicar su nombre junto con una lista de argumentos entre paréntesis. En nuestro programa la primera llamada es:

```
DibujarLogo(papel, miLápiz, 10, 20);
```

Esta instrucción tiene dos efectos:

- Los valores de los argumentos se transfieren al método de manera automática. Más adelante hablaremos sobre esto con mayor detalle.
- El programa salta al cuerpo del método (la instrucción después del encabezado) y ejecuta las instrucciones. Cuando termina con todas las instrucciones y llega al carácter }, la ejecución continúa en el punto desde el que se hizo la llamada al método.

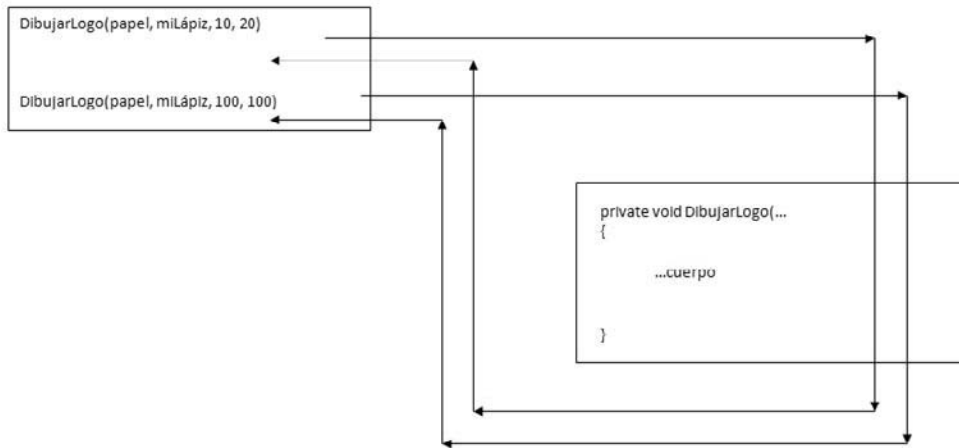


Figura 5.3 Ruta de ejecución de dos llamadas.

Después se lleva a cabo la segunda llamada:

```
DibujarLogo(papel, miLápiz, 100, 100);
```

En la figura 5.3 se ilustra esto. Son dos llamadas que producen dos logotipos.

## ● Cómo pasar argumentos

Es imprescindible comprender lo mejor posible cómo se transfieren (pasan) los argumentos a los métodos. En nuestro ejemplo el concepto se muestra en las siguientes líneas:

```
DibujarLogo(papel, miLápiz, 10, 20);
private void DibujarLogo(Graphics áreaDibujo,
    Pen lápizAUsar,
    int posX,
    int posY)
```

El área en la que debemos enfocarnos está constituida por las dos listas de elementos que se hallan entre paréntesis. En una llamada los elementos se denominan *argumentos*. En el encabezado del método los elementos se denominan *parámetros*. Para aclarar esta situación extraigamos los parámetros y los argumentos:

argumentos:	papel	miLápiz	10	20
parámetros:	áreaDibujo	lápizAUsar	posX	posY

Recordemos la comparación que hicimos de una variable con una caja. Dentro del método hay un conjunto de cajas vacías (los parámetros) que esperan la transferencia de los valores de los argumentos. Después de la transferencia tenemos la situación que se muestra en la Figura 5.4. No contamos con valores numéricos para pasarlos al área de dibujo y el lápiz, por lo que nos enfocaremos en cómo se transfieren las coordenadas.

La transferencia se realiza de izquierda a derecha. La llamada debe proporcionar el número y tipo correctos de cada argumento. Si el que hace la llamada (el usuario) recibe accidentalmente los argumentos en el orden incorrecto, el proceso de la transferencia no los regresará a su orden correcto.



**Figura 5.4** Transferencia de los argumentos a los parámetros.

Cuando el método `DibujarLogo` se ejecuta, estos valores controlan el proceso de dibujo. Aunque en este ejemplo invocamos el método con números, también podemos utilizar expresiones (es decir, incluir variables y cálculos), como en el siguiente ejemplo:

```
int x = 6;
DibujarLogo(papel, miLápiz, 20 + 3, 3 * 2 + 1); // 23 y 7
DibujarLogo(papel, miLápiz, x * 4, 20); // 24 y 20
```

En C# hay dos formas de pasar elementos a un método: por valor (como en los ejemplos anteriores) y por referencia. Más adelante en este capítulo veremos cómo pasar elementos por referencia.

### PRÁCTICA DE AUTOEVALUACIÓN

#### 5.1 ¿En qué posición se dibujarán los logotipos si se utiliza el siguiente código?

```
int a = 10;
int b = 20;
DibujarLogo(papel, miLápiz, a, b);
DibujarLogo(papel, miLápiz, b + a, b - a);
DibujarLogo(papel, miLápiz, b + a - 3, b + a - 4);
```

## ● Parámetros y argumentos

En el análisis que estamos llevando a cabo están involucradas dos listas entre paréntesis, y es importante aclarar el propósito de cada una de ellas:

- El programador que escribe el método debe elegir cuáles son los elementos que éste solicitará por medio de parámetros. En el método `DibujarLogo` las medidas de los cuadrados anidados siempre se establecen en 20, 40 y 60, de manera que la instancia que invoca el método no necesita suministrar esos datos. Sin embargo, tal vez quien haga la llamada quiera variar la posición del logotipo, utilizar un lápiz distinto o incluso dibujarlo en un componente distinto (como un botón). Estos elementos se han convertido en parámetros.
- El escritor del método debe elegir el nombre de cada parámetro. Si se utilizan nombres similares en otros métodos no hay problema alguno, pues cada uno de ellos tiene una copia propia de sus parámetros. En otras palabras, el escritor tiene la libertad de elegir cualquier nombre.
- Se debe proporcionar el tipo de cada parámetro; esta información debe ir antes del nombre del mismo. Los tipos dependen del método en particular. Se utiliza una coma para separar los parámetros entre sí. En el encabezado de `DibujarLogo` puede comprobar este acomodo.
- Quien hace la llamada debe proveer una lista de argumentos entre paréntesis, y éstos tendrán que ser del tipo correcto y estar en el orden adecuado para el método.

Los dos beneficios que conlleva la utilización de un método para dibujar el logotipo son: evitamos duplicar las tres instrucciones `DrawRectangle` cuando se requieren varios logos, y al dar un nombre a esta tarea podemos ser decididamente más creativos.

Por último, sabemos que tal vez desee aplicar las habilidades de programación que ha aprendido aquí a otros lenguajes, pero debe saber que aun cuando los conceptos son similares la terminología podría ser distinta; por ejemplo, en muchos lenguajes quien hace la llamada provee los “parámetros actuales”, y la declaración del método tiene “parámetros formales”.

### PRÁCTICAS DE AUTOEVALUACIÓN

**5.2** Explique cuál es el error en estas llamadas:

```
DibujarLogo(papel, miLápiz, 50, "10");
DibujarLogo(miLápiz, papel, 50, 10);
DibujarLogo(papel, miLápiz, 10);
```

**5.3** La siguiente es la llamada a un método:

```
SóloHazlo("Naranjas");
```

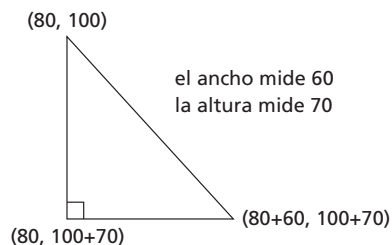
y éste es el código del método:

```
private void SóloHazlo(string fruta)
{
    MessageBox.Show(fruta);
}
```

¿Qué ocurre cuando se invoca este método?

## ● Un método para dibujar triángulos

Con el propósito de presentar más características de los métodos crearemos uno más útil y le llamaremos `DibujarTriángulo`. En vista de que *escribiremos* el método en vez de utilizar uno predefinido, podemos elegir qué tipo de triángulo se va a dibujar y los argumentos que deseamos que proporcione quien haga la llamada. En este caso dibujaremos un triángulo rectángulo con el ángulo recto a la derecha, como se muestra en la Figura 5.5.



**Figura 5.5** Cálculo de las coordenadas de un triángulo.

Para elegir los argumentos tenemos varias posibilidades: por ejemplo, podríamos requerir que quien haga la llamada proporcione las coordenadas de las tres esquinas. Sin embargo, optaremos por usar los siguientes argumentos:

- el área de dibujo y el lápiz, como en el método anterior;
- las coordenadas del punto superior del triángulo;
- el ancho del triángulo;
- la altura del triángulo.

Otra manera de considerar estas coordenadas consiste en hacer que especifiquen la posición de un rectángulo circundante para nuestro triángulo recto.

Podemos dibujar las líneas en cualquier orden. Empecemos por examinar el proceso de dibujo con números. A manera de ejemplo dibujaremos un triángulo con la esquina superior en (80, 100), con un ancho de 60 y una altura de 70. En la Figura 5.5 se muestran los cálculos. El proceso es el siguiente:

- Dibujar de (80, 100) hasta (80, 100+70). Recuerde que la coordenada y se incrementa hacia abajo.
- Dibujar de (80, 100+70) hasta (80+60, 100+70).
- Dibujar en diagonal desde la esquina superior (80, 100) hasta (80+60, 100+70).

Asegúrese de poder seguir el proceso anterior; tal vez sea conveniente que primero lo dibuje en papel.

Observe que en nuestra explicación no simplificamos los cálculos: dejamos 100+70 en su forma original, en vez de usar 170. Al llegar a la codificación, la posición del triángulo y su tamaño se pasarán como argumentos separados.

El siguiente es el código completo del programa Método triángulo. Este programa contiene un método llamado `DibujarTriángulo`, y también incluye el método `DibujarLogo` para ilustrar que un programa puede contener muchos métodos.

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    Pen miLápiz = new Pen(Color.Black);
    DibujarLogo(papel, miLápiz, 10, 20);
    DibujarLogo(papel, miLápiz, 100, 100);
    DibujarTriángulo(papel, miLápiz, 100, 10, 40, 40);
    DibujarTriángulo(papel, miLápiz, 10, 100, 20, 60);
}

private void DibujarLogo(Graphics áreaDibujo,
    Pen lápizAUsar,
    int posX,
    int posY)
{
    áreaDibujo.DrawRectangle(lápizAUsar, posX, posY, 60, 60);
    áreaDibujo.DrawRectangle(lápizAUsar, posX, posY, 40, 40);
    áreaDibujo.DrawRectangle(lápizAUsar, posX, posY, 20, 20);
}
```



```
private void DibujarTriángulo(Graphics áreaDibujo,
    Pen lápizAUsar,
    int lugarX,
    int lugarY,
    int ancho,
    int altura)
{
    áreaDibujo.DrawLine(lápizAUsar, lugarX, lugarY,
        lugarX, lugarY + altura);
    áreaDibujo.DrawLine(lápizAUsar, lugarX,
        lugarY + altura,
        lugarX + ancho, lugarY + altura);
    áreaDibujo.DrawLine(lápizAUsar, lugarX, lugarY,
        lugarX + ancho, lugarY + altura);
}
```

Nuestro programa tiene un botón y un cuadro de imagen. Haga clic en el botón y se dibujarán dos logotipos y dos triángulos. En la Figura 5.6 se muestra el resultado.

Veamos algunos detalles sobre la codificación del método `DibujarTriángulo`:

- Pudimos llamarlo `Triángulo`, o incluso `DibujarCosa`, pero elegimos denominarlo `DibujarTriángulo` porque este nombre se ajusta al estilo de los métodos de la biblioteca.

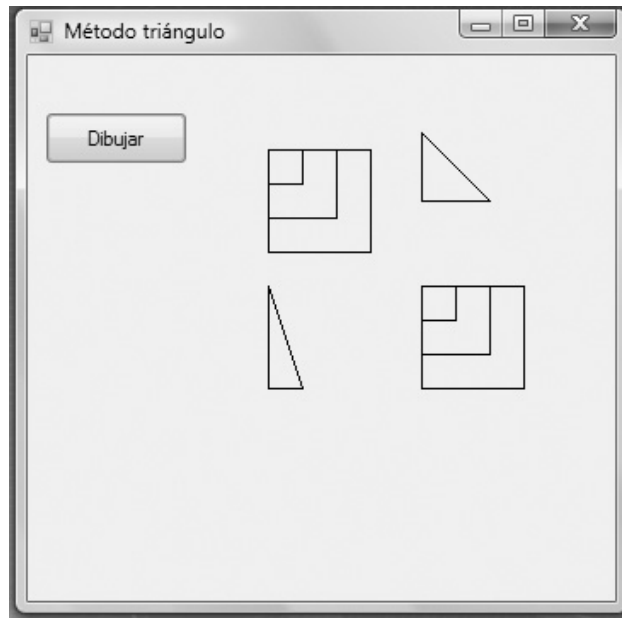


Figura 5.6 El programa Método triángulo.

- Nosotros escogimos los nombres de los parámetros: `áreaDibujo`, `lápizAUsar`, `lugarX`, `lugarY`, `ancho` y `altura`.
- El orden de los parámetros también está bajo nuestro control. Si quisiéramos, podríamos volver a codificar el método para requerir la altura antes del ancho (pusimos el ancho primero debido a que muchos de los métodos de la biblioteca de C# utilizan ese orden).

El resultado es que ahora tenemos nuestro triángulo. Utilizaremos el programa para analizar las variables locales, y también para demostrar que puede ser la base de métodos más poderosos.

## ● Variables locales

Dé un vistazo a la siguiente versión modificada del método `DibujarTriángulo`, a la que hemos llamado `DibujarTriángulo2`:

```
private void DibujarTriángulo2(Graphics áreaDibujo,
    Pen lápizAUsar,
    int lugarX,
    int lugarY,
    int ancho,
    int altura)
{
    int esquinaDerechaX, esquinaDerechaY;
    esquinaDerechaX = lugarX + ancho;
    esquinaDerechaY = lugarY + altura;

    áreaDibujo.DrawLine(lápizAUsar, lugarX, lugarY,
        lugarX, esquinaDerechaY);
    áreaDibujo.DrawLine(lápizAUsar, lugarX, esquinaDerechaY,
        esquinaDerechaX, esquinaDerechaY);
    áreaDibujo.DrawLine(lápizAUsar, lugarX, lugarY,
        esquinaDerechaX, esquinaDerechaY);
}
```

Este método se invoca de la misma forma que `DibujarTriángulo`, pero internamente utiliza dos variables llamadas `esquinaDerechaX` y `esquinaDerechaY`, las cuales fueron introducidas para simplificar los cálculos. Vea cómo se utilizan para hacer referencia al punto del triángulo que está más a la derecha. Estas variables sólo existen dentro de `DibujarTriángulo2`. Son locales para el método (de acuerdo con la terminología de programación, se dice que tienen *alcance* local). Si existen variables del mismo nombre dentro de otros métodos no hay conflicto, ya que cada método utiliza su propia copia. Otra manera de entender esto es que cuando distintos programadores creen métodos podrán inventar las variables locales sin tener que comparar sus nombres con los utilizados por sus colegas.

La función que desempeñan las variables locales consiste en ayudar al método a realizar su trabajo, sin importar cuál sea éste. Las variables tienen un alcance limitado, ya que están restringidas a su propio método. Su existencia es temporal: se crean al momento de invocar el método, y se destruyen cuando el método termina de ejecutarse.

## ● Conflictos de nombres

Ya hemos visto que en C# el creador de un método tiene la libertad de elegir nombres apropiados para las variables locales y los parámetros. ¿Pero qué ocurre si se escogen nombres que estén en conflicto con otras variables?

Podríamos tener lo siguiente:

```
private void MétodoUno(int x, int y)
{
    int z = 0;
    // código...
}

private void MétodoDos(int z, int x)
{
    int w = 1;
    // código...
}
```

Suponga que dos personas distintas escribieron estos métodos. **MétodoUno** tiene los parámetros **x** y **y**; además declara una variable tipo entero llamada **z**. Estos tres elementos son locales para **MétodoUno**. En **MétodoDos** el programador ejerce su derecho a nombrar los elementos locales, y opta por usar **z**, **x** y **w**. El conflicto de nombres respecto de la **x** (y la **z**) no representa un problema, ya que C# considera que la **x** de **MétodoUno** y la **x** de **MétodoDos** son distintas.

### PRÁCTICA DE AUTOEVALUACIÓN

**5.4** Considere la siguiente llamada a un método:

```
int a = 3;
int b = 8;
HacerAlgo(a, b);
MessageBox.Show(Convert.ToString(a));
```

considere también el siguiente método:

```
private void HacerAlgo(int x, int y)
{
    int a = 0;
    a = x + y;
}
```

¿Qué se despliega en el cuadro de mensaje?

Veamos un resumen de las herramientas para trabajar con métodos que hemos analizado hasta el momento (más adelante incluiremos la instrucción **return** y el paso de parámetros por referencia).

- La forma general de la declaración de un método cuyo propósito no es calcular un resultado y al que los argumentos le son transferidos por valor es:

```
private void UnNombre(lista de parámetros)
{
    cuerpo
}
```

El programador elige el nombre del método.

- La lista de parámetros es una lista de tipos y nombres. Si un método no necesita argumentos utilizamos paréntesis vacíos inmediatamente después de declararlo, y también para la lista de argumentos al momento de invocarlo.

```
private void MiMétodo()
{
    cuerpo
}
```

y la llamada al método sin argumentos tendría esta forma:

```
MiMétodo();
```

- Una clase puede contener cualquier número de métodos, en el orden que se desee. Los programas que empleamos como ejemplo en este capítulo sólo incluyen una clase. En esencia su distribución es:

```
public class Formal
{
    private void UnNombre(lista de parámetros...)
    {
        cuerpo
    }
    private void OtroNombre(lista de parámetros...)
    {
        cuerpo
    }
}
```

En el capítulo 10 veremos cómo usar la palabra clave `class`. Por ahora basta con tener en cuenta que una clase puede agrupar una serie de métodos.

## ● Métodos para manejar eventos

---

Una clase puede contener un conjunto de métodos. Algunos de ellos son escritos por el propio programador (como hicimos con `DibujarLogo`), y son invocados de manera explícita. Sin embargo, hay otros que C# crea automáticamente, como en el siguiente ejemplo:

```
private void button1_Click
```

¿Cuándo se invoca este método? La respuesta es que el sistema de C# dirige todos los eventos (como el clic en botones o del ratón, etc.) hacia el método de evento apropiado, siempre y cuando éste exista. Por lo general nosotros nunca los invocamos.

## ● La instrucción `return` y los resultados

En nuestros ejemplos anteriores de argumentos y parámetros pasamos valores *hacia* los métodos, para que éstos los utilizaran. Sin embargo, con frecuencia es necesario codificar métodos que realicen un cálculo y envíen un resultado al resto del programa, de manera que pueda emplearlo en cálculos posteriores. En estos casos podemos utilizar la instrucción `return`. Veamos un método que calcula el área de un rectángulo, dados sus dos lados como argumentos de entrada. El siguiente es el código del programa completo, llamado Método Área:

```
private void button1_Click(object sender, EventArgs e)
{
    int a;
    a = ÁreaRectángulo(10, 20);
}

private int ÁreaRectángulo(int longitud, int ancho)
{
    int área;
    área = longitud * ancho;
    return área;
}
```

Este ejemplo incluye varias nuevas características relacionadas entre sí.

Considere el encabezado del método:

```
private int ÁreaRectángulo(int longitud, int ancho)
```

En vez de `void` especificamos el tipo de elemento que el método debe regresar a la instancia que lo invocó. Como en este caso estamos multiplicando dos valores `int`, la respuesta también es de tipo `int`.

La elección del tipo de este elemento depende del problema. Por ejemplo, el resultado que estamos buscando podría ser un entero o una cadena de caracteres, pero también podría ser un objeto más complicado, como un cuadro de imagen o un botón. El programador que escribe el método elige el tipo de valor que se devolverá.

Para devolver un valor como resultado del método utilizamos la instrucción `return` de la siguiente manera:

```
return expresión;
```

La expresión (como siempre) puede ser un número, una variable o un cálculo (o incluso la llamada a un método), pero es necesario que sea del tipo correcto, según lo especificado en la declaración del método (su encabezado). Además, la instrucción `return` hace que el método actual deje de ejecutarse, y regresa de inmediato al lugar en el que se encontraba dentro del método que hizo la llamada. Ahora veamos cómo se puede invocar un método que devuelve un resultado.

La siguiente es una manera de no invocar dicho método. No debe utilizarse como una instrucción completa; por ejemplo:

```
ÁreaRectángulo(10, 20);    //no
```

En lugar de ello, el programador debe asegurarse de “utilizar” el valor devuelto. Para comprender cómo devolver un valor imagine que la llamada al método (el nombre y la lista de argumentos) se elimina, y se sustituye por el resultado devuelto. Si el código resultante tiene sentido, C# le permitirá realizar dicha llamada. Vea este ejemplo:

```
respuesta = ÁreaRectángulo(30, 40);
```

El resultado es 1200, y si sustituimos la llamada por este resultado, obtenemos lo siguiente:

```
respuesta = 1200;
```

Este código de C# es válido, pero si utilizamos:

```
ÁreaRectángulo(30, 40);
```

al sustituir el resultado devuelto se produciría una instrucción de C# que constaría sólo de un número:

```
1200;
```

lo cual no tiene significado (aunque en sentido estricto el compilador de C# permitirá que se ejecute la llamada a `ÁreaRectángulo`. Sin embargo, no tiene caso descartar el resultado devuelto por un método cuyo propósito principal es devolver dicho resultado).

Las siguientes líneas de código son formas válidas en las que podríamos “utilizar” el resultado:

```
private button2_Click(object sender, EventArgs e)
{
    int n;
    n = ÁreaRectángulo(10, 20);
    MessageBox.Show("el área mide " +
        Convert.ToString(ÁreaRectángulo(3, 4)));
    n = ÁreaRectángulo(10, 20) * ÁreaRectángulo(7, 8);
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

**5.5** Utilice lápiz y papel para trabajar con las instrucciones anteriores; sustituya los resultados por las llamadas.

Para completar nuestro análisis sobre la instrucción `return`, cabe mencionar que podemos utilizarla con métodos `void`. En ese caso debemos emplear `return` sin especificar un resultado, como en el ejemplo siguiente:

```
private void Demo(int n)
{
    // hacer algo
    return;
    // hacer otra cosa
}
```

Esta característica puede aprovecharse cuando queremos que el método termine en una instrucción que no sea la última.

Veamos una manera alternativa de codificar nuestro ejemplo del área:

```
private int ÁreaRectángulo2(int longitud, int ancho)
{
    return longitud * ancho;
}
```

Debido a que podemos utilizar `return` con expresiones, hemos omitido la variable `área` en `ÁreaRectángulo2`.

Estas reducciones al tamaño del programa no siempre son beneficiosas, ya que sacrificar el uso de nombres representativos puede demeritar la claridad y, por ende, exigir más tiempo de depuración y prueba.

### PRÁCTICA DE AUTOEVALUACIÓN

**5.6** El siguiente método se llama `Doble` y devuelve el doble del valor de su argumento `int`.

```
private int Doble(int n)
{
    return 2 * n;
}
```

Dadas las siguientes llamadas al método:

```
int n = 3;
int r;
r = Doble(n);
r = Doble(n + 1);
r = Doble(n) + 1;
r = Doble(3 + 2 * n);
r = Doble(Doble(n));
r = Doble(Doble(n + 1));
r = Doble(Doble(n) + 1);
r = Doble(Doble(Doble(n)));
```

Indique el valor devuelto por cada llamada.

## ● Construcción de métodos a partir de otros métodos

Como ejemplo de métodos que utilizan otros métodos, a continuación crearemos un método que dibuja una casa sencilla con una sección transversal, tal como se muestra en la Figura 5.7. La altura del techo es la misma que la altura de las paredes, y el ancho de la pared es el mismo que el ancho del techo. Utilizaremos los siguientes argumentos `int`:

- la posición horizontal del punto superior derecho del techo;
- la posición vertical del punto superior derecho del techo;

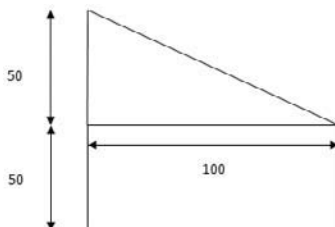


Figura 5.7 El ancho de esta casa mide 100, y la altura de su techo mide 50.

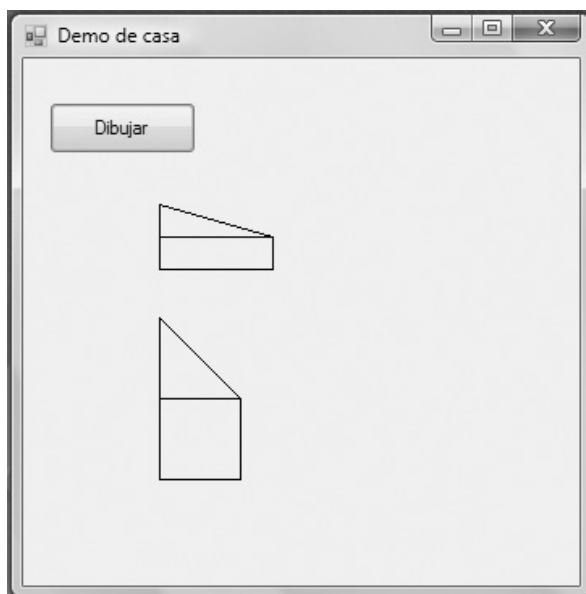


Figura 5.8 El programa Demo de casa.

- el ancho de la casa. El triángulo que representa el techo y el rectángulo que simula las paredes tienen el mismo ancho;
- la altura del techo (excluyendo la pared).

Utilizaremos el método `DrawRectangle` de la biblioteca de C#, y también nuestro propio método `DibujarTriángulo`.

El siguiente es el código del programa, cuyos resultados se muestran en la Figura 5.8.

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    Pen miLápiz = new Pen(Color.Black);
    DibujarCasa(papel, miLápiz, 10, 20, 70, 20);
    DibujarCasa(papel, miLápiz, 10, 90, 50, 50);
}
```



```
private void DibujarCasa(Graphics áreaDibujo,
    Pen lápizAUsar,
    int techoSupX,
    int techoSupY,
    int ancho,
    int altura)
{
    DibujarTriángulo(áreaDibujo, lápizAUsar, techoSupX, techoSupY,
        ancho, altura);
    áreaDibujo.DrawRectangle(lápizAUsar, techoSupX,
        techoSupY + altura, ancho, altura);
}

private void DibujarTriángulo(Graphics áreaDibujo,
    Pen lápizAUsar,
    int lugarX,
    int lugarY,
    int ancho,
    int altura)
{
    áreaDibujo.DrawLine(lápizAUsar, lugarX, lugarY,
        lugarX, lugarY + altura);
    áreaDibujo.DrawLine(lápizAUsar, lugarX,
        lugarY + altura,
        lugarX + ancho, lugarY + altura);
    áreaDibujo.DrawLine(lápizAUsar, lugarX, lugarY,
        lugarX + ancho, lugarY + altura);
}
```

El programa es bastante fácil de comprender si consideramos que:

- Los métodos regresan al punto desde el cual se invocaron, por lo cual:
  - button1\_Click invoca a DibujarCasa;
  - DibujarCasa invoca a DrawRectangle;
  - DibujarCasa invoca a DibujarTriángulo;
  - DibujarTriángulo invoca a DrawLine (tres veces).
- Los argumentos pueden ser expresiones, por lo que se evalúa `lugarY + altura`, y el resultado se pasa a `DrawLine`.
- Las variables `ancho` y `altura` de `DibujarCasa`, y las variables `ancho` y `altura` de `DibujarTriángulo` son totalmente distintas. Sus valores se almacenan en diferentes lugares.

En este ejemplo podemos ver que lo que hubiera podido ser un programa más grande se ha escrito como un programa corto, dividido en métodos y con nombres representativos. Esto ilustra el poder que se obtiene al utilizar métodos.

## ● Transferencia de argumentos por referencia

---

Hasta ahora hemos utilizado el concepto de pasar (o transferir) argumentos por valor, ya sea mediante un método que devuelve un valor, o a través de un método *void* que no hace devolución alguna. Esto está bien para la mayoría de las situaciones, ¿pero qué pasa si es necesario que un método devuelva más de un resultado? Considere la siguiente situación:

Dada una cantidad de centavos, escriba un método para calcular el número entero de dólares equivalente, y el número de centavos restantes. Tenemos una entrada y dos resultados.

Antes de conocer la metodología utilizada en C# para devolver varios resultados, es preciso que analicemos con más detalle la naturaleza del paso de argumentos. Anteriormente mencionamos que pasamos *valores* como argumentos. Esto parece obvio; ¿qué otra cosa podríamos hacer? Pues bien, de hecho C# también nos permite pasar argumentos por *referencia*, y podemos utilizar esta herramienta para devolver cualquier número de resultados de un método.

He aquí una analogía para ilustrar el paso de argumentos por referencia: imagine que está colaborando con algunos colegas en la elaboración de un informe escrito, y uno de ellos le pide cierto documento. Hay dos formas en las que puede pasar ese documento a su colega:

- Fotocopiarlo.
- Puede decirle: “Ah claro, ese documento. Búscalo en la cuarta repisa de arriba hacia abajo del archivero. Ahí está”.

La analogía:

- El documento es un argumento.
- Su colega es un método al que usted está invocando, y al que debe pasar un argumento.
- El proceso de pasar una fotocopia del documento representaría la transferencia “por valor”.
- Decir a su colega en dónde está guardado el documento original sería una “transferencia por referencia”.

Estas formas de pasar argumentos involucran dos puntos importantes:

- Es más seguro pasar una copia de los datos (transferencia por valor), pues de esa manera usted se queda con el elemento original. Cualquier modificación que realice su colega *no* afectará la copia que usted conserva.
- Es más rápido pasar la ubicación de los datos (transferencia por referencia). De hecho los datos no se copian ni desplazan físicamente, y su colega puede realizar modificaciones sin tener que sacar el documento de la habitación. Pero recuerde que sólo hay una copia del elemento. Su colega tiene el mismo poder que usted para modificar esa copia única. Algunas veces esto será conveniente, pero otras no.

Teniendo en mente los conceptos de pasar por valor y pasar por referencia, veamos ahora cómo se organiza la memoria de acceso aleatorio (RAM) de la computadora. La RAM consiste en millones de cajas de almacenamiento, conocidas como ubicaciones de memoria. Cada ubicación tiene una dirección, algo similar a la nomenclatura en una calle. Cada variable que creamos se almacena en un lugar específico de la memoria. En otras palabras, cada variable se asocia con una dirección.

Hemos llegado al punto en que veremos cómo pasar los argumentos. Si deseamos pasar una variable a un método hay dos opciones:

- pasar una copia del valor actual;
- o pasar la dirección. En la jerga técnica de C# se dice que pasamos una referencia a la variable. Cuando el método conoce la ubicación de una variable sabe en qué lugar de la memoria debe buscar. En algunos otros lenguajes a este tipo de referencias se les conoce como *apuntadores*. Como veremos más adelante, en C# hay dos estilos para pasar referencias: utilizando las palabras clave `ref` o `out`.

#### PRÁCTICA DE AUTOEVALUACIÓN

**5.7** Imagine que tenemos una gran cantidad de variables almacenadas en RAM. Si conocemos el valor de una variable, ¿es posible averiguar en dónde está almacenada? Explique su respuesta?

### ● Los argumentos out y ref

Las palabras clave `out` y `ref` permiten que el programador especifique con mucha precisión de qué manera el método que recibe los argumentos puede acceder a éstos y modificarlos. Hay dos posibilidades:

- El método invocado *modifica* el valor existente de un argumento en cierta forma. A esto se le conoce comúnmente como “actualizar”. Para ello es preciso que el método acceda al valor actual del argumento. En este caso utilizamos la palabra clave `ref`.
- El método invocado coloca un valor completamente nuevo en un argumento, por lo cual no necesita acceder a su valor actual. En este caso utilizamos la palabra clave `out`.

Veamos la relación entre esto y la analogía del informe que vimos antes:

- Si queremos que nuestro colega (un método) modifique un informe existente, debemos utilizar `ref`.
- Si queremos que nuestro colega cree un informe completamente nuevo, le proporcionamos un informe vacío para que lo complete. En este caso debemos usar `out`.

A continuación revisaremos dos ejemplos que ilustran el uso de `out` y `ref`.

### ● Un ejemplo con out

Analicemos nuevamente uno de los problemas que planteamos previamente: un método que convierte un número de centavos en un número entero de dólares y los centavos sobrantes. Para resolver este problema se requiere codificar un método con una entrada y dos resultados. En la Figura 5.9 se muestra el programa Método Dólares en ejecución; el código correspondiente, que utiliza un cuadro de texto para obtener el número de centavos junto con etiquetas para mostrar los resultados, aparece a continuación. Utilizamos una GUI similar en el capítulo 4, cuando realizamos la misma tarea sin utilizar un método para realizar el cálculo.

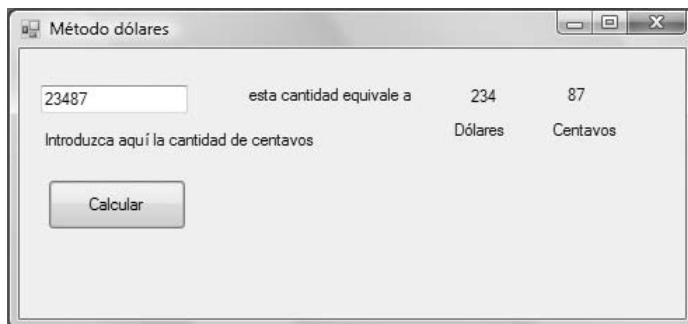


Figura 5.9 El programa Método Dólares.

```
private void button1_Click(object sender, EventArgs e)
{
    int centavosOriginales, dólaresEnteros = 0, centavosSobrantes = 0;
    centavosOriginales = Convert.ToInt32(textBox1.Text);
    DólaresYCentavos(centavosOriginales, out dólaresEnteros,
        out centavosSobrantes);
    dólaresEtiqueta.Text = Convert.ToString(dólaresEnteros);
    centavosEtiqueta.Text = Convert.ToString(centavosSobrantes);
}

private void DólaresYCentavos(int totalCentavos,
    out int dólares, out int centavosSobrantes)
{
    dólares = totalCentavos / 100;
    centavosSobrantes = totalCentavos % 100;
}
```

Los valores de las propiedades se muestran a continuación:

Control	Propiedad	Valor
button1	Text	Calcular
textBox1	Text	(vacía)
Label (dólares)	Text	(vacía)
	Name	dólaresEtiqueta
Label (centavos)	Text	(vacía)
	Name	centavosEtiqueta

He aquí algunas observaciones sobre el programa anterior:

- Elegimos el nombre `DólaresYCentavos` para el método.
- El método implica la devolución de dos resultados, por lo que no podemos usar la instrucción `return`.
- El parámetro `totalCentavos` se pasa por valor. Al hacer una transferencia por valor nos aseguramos de que el método no pueda modificar el valor original y resulte, por lo tanto, más seguro.

- Los parámetros `dólares` y `centavosSobrantes` se pasan por referencia, utilizando la palabra clave `out`. Los valores que el método colocará en estas variables no se basan en aquellos que pudieran tener. Son, en efecto, cajas vacías y estamos informando a `DólaresYCentavos` sobre su ubicación.
- Al declarar un método no se pone nada antes de los parámetros que se pasan por valor. En el caso de los parámetros `out` hay que incluir la palabra clave `out` antes de su nombre. Por ejemplo:

```
private void DólaresYCentavos(int totalCentavos,
    out int dólares, out int centavosSobrantes)
```

Al invocar un método debemos colocar la palabra `out` antes de cualquier argumento `out`, como en el siguiente ejemplo:

```
DólaresYCentavos(centavosOriginales, out dólaresEnteros,
    out centavosSobrantes);
```

Observe que no se hace mención del tipo de una variable al invocar un método. Esto sólo se indica en la declaración del método.

- Cuando el método asigna un nuevo valor a `centavosSobrantes` en realidad está usando la variable `centavosSobrantes`, declarada en el método `button1_Click`.
- Como siempre, el escritor del método tiene la libertad de elegir los nombres de los argumentos. No es necesario que estén relacionados con los nombres usados en otros métodos.
- Si el método intenta utilizar el valor actual de un parámetro `out` antes de que se haya guardado un nuevo valor en él, se producirá un error de compilación. Éste es un caso de ejemplo: colocamos la siguiente declaración justo después del carácter de apertura “{” del método `DólaresYCentavos`.

```
int temp = dólares;
```

Aquí se ha especificado `dólares` como `out`, y C# toma esto muy en serio. El programa no podrá acceder al valor actual de dicho parámetro. Si queremos que lo haga tendremos que utilizar `ref`, como veremos a continuación.

## PRÁCTICAS DE AUTOEVALUACIÓN

- 5.8** Suponga que un cajero automático sólo puede dispensar billetes de 10 dólares y de 1 dólar. Escriba un método que calcule el número de billetes de 10 dólares y de 1 dólar para cualquier cantidad de dinero solicitada. He aquí un ejemplo de la llamada a ese método:

```
CalcularBilletes(cantidad, out billetesDiez, out billetesUno);
```

- 5.9** ¿Funcionaría el método `DólaresYCentavos` correctamente si modificáramos su código de la siguiente manera?

```
private void DolaresYCentavos2(int a, out int b, out int c)
{
    b = a / 100;
    c = a % 100;
}
```

## ● Un ejemplo con `ref`

---

Veamos ahora un programa (Tamaño de panes) que calcula el área de una serie de panes rectangulares. El método suma 2 al ancho y al largo del pan original, y después utiliza un cuadro de mensaje para desplegar las nuevas áreas del mismo. Observe que las nuevas medidas se basan en las anteriores (utilizamos sus valores actuales), por lo que debemos utilizar `ref` en vez de `out`. He aquí el código:

```
private void button1_Click(object sender, EventArgs e)
{
    int anchoPan = 8, largoPan = 6;
    IncrementarTamaño(ref anchoPan, ref largoPan);
    IncrementarTamaño(ref anchoPan, ref largoPan);
    IncrementarTamaño(ref anchoPan, ref largoPan);
}

private void IncrementarTamaño(ref int ancho, ref int largo)
{
    int área;
    ancho = ancho + 2;
    largo = largo + 2;
    área = ancho * largo;
    MessageBox.Show("Tamaño del pan: " + Convert.ToString(ancho) +
        " por " + Convert.ToString(largo) + ". El área del pan mide " +
        Convert.ToString(área));
}
```

La interfaz de usuario consiste sólo de un botón para iniciar el cálculo, por lo que no la mostramos aquí. Los tres cuadros de mensaje resultantes muestran lo siguiente:

```
Tamaño del pan: 10 por 8. El área del pan mide 80
Tamaño del pan: 12 por 10. El área del pan mide 120
Tamaño del pan: 14 por 12. El área del pan mide 168
```

A continuación, algunas observaciones respecto de este programa:

- Elegimos `IncrementarTamaño` como nombre del método.
- El método modifica los valores existentes de `anchoPan` y `largoPan`, los cuales se declaran e inicializan en el método `button1_Click`.
- Al sumar 2 a `ancho` y a `largo` dentro de `IncrementarTamaño`, los nuevos valores aparecen en `anchoPan` y `largoPan` del método `button1_Click`. Es como si `ancho` y `largo` representaran a `anchoPan` y `largoPan`.
- Al declarar un método anteponeamos la palabra clave `ref` a cualquier parámetro por referencia. Por ejemplo:

```
private void IncrementarTamaño(ref int ancho, ref int largo)
```

Al invocar un método colocamos la palabra clave `ref` antes de cualquier argumento `ref`, como en el siguiente ejemplo:

```
IncrementarTamaño(ref anchoPan, ref largoPan);
```

Tome en cuenta que no mencionamos el tipo de una variable en la llamada. El tipo sólo se indica en la declaración del método.

- El escritor del método tiene la libertad de elegir los nombres de los argumentos. En este caso escogimos nombres distintos de los que se utilizan en el método `button1_Click`, pero podríamos haber empleado `anchoPan` y `largoPan` como nombres de los parámetros dentro del método `IncrementarTamaño`. Si lo hubiéramos hecho así el programa se habría ejecutado exactamente igual, debido a que los nombres de los parámetros son locales para el método en el que se declaran.

## PRÁCTICA DE AUTOEVALUACIÓN

**5.10** El código siguiente es la llamada a un método:

```
int x = 4;
int y = 9;
HacerTarea(x, ref y);
```

y éste es el método:

```
private void HacerTarea(int a, ref int b)
{
    a = a + 1;
    b = b + 1;
}
```

¿Cuáles son los valores resultantes de `x` y `y`?

## ● Un método de intercambio con `ref`

Un ejemplo clásico del uso de argumentos es el código de un método que intercambia los valores de dos variables. Primero veamos cómo sería el código sin hacer uso de un método:

```
aCopia = a;
a = b;
b = aCopia;
```

Observe que, si utilizamos:

```
a = b;
b = a;
```

tanto `a` como `b` terminan con el valor original de `b`.

Por lo tanto, en vez de usar código que sólo funcione para las variables `a` y `b`, lo que haremos es agruparlo en forma de un método que funcione para cualquier variable. Hay dos argumentos; el

método necesita acceder a sus valores originales y modificarlos. Se requiere el uso de `ref` en vez de `out` (si utilizamos `out` el método no podrá usar los valores originales de los argumentos). He aquí el código:

```
private void Intercambiar(ref int a, ref int b);
{
    int aCopia;
    aCopia = a;
    a = b;
    b = aCopia;
}
```

A continuación algunos ejemplos de cómo llamar al método `Intercambiar`:

```
int a = 6;
int b = 8;
int c = 20;

Intercambiar(ref a, ref b);
Intercambiar(ref a, ref c);
```

### PRÁCTICA DE AUTOEVALUACIÓN

---

**5.11** Explique cuál es el error en las siguientes llamadas al método `Intercambiar`.

```
int x = 4, y = 5;
Intercambiar(x, y);
Intercambiar(ref x, ref 6);
```

Resumiendo lo que hemos visto sobre los métodos:

- pasar argumentos por valor;
- devolver un solo valor de un método mediante `return`, lo cual es suficiente para la mayoría de los casos;
- pasar referencias con `out` y `ref`, lo cual se utiliza pocas veces.

## ● La palabra clave `this` y los objetos

---

Probablemente esté leyendo este libro debido a que C# es un lenguaje orientado a objetos, pero tal vez se esté preguntando por qué no hemos mencionado los objetos en este capítulo. La verdad es que los métodos y los objetos tienen una conexión vital. Al ejecutar programas pequeños realizados en C# estamos ejecutando una instancia de una clase; es decir, un objeto. Este objeto contiene métodos (como `Intercambiar`) y propiedades (tema que no hemos abordado todavía).

Cuando un objeto invoca un método que está declarado en su interior, podemos simplificar la llamada utilizando:

```
Intercambiar(ref a, ref b);
```



o utilizar la notación de objetos completa, como en el siguiente ejemplo:

```
this.Intercambiar(ref a, ref b);
```

`this` es una palabra clave de C#, y representa el objeto actual en ejecución. Por lo tanto, a lo largo de todo el capítulo usted ha estado utilizando la programación orientada a objetos sin darse cuenta de ello. He aquí algunos ejemplos:

```
Intercambiar(ref a, ref b);

// funciona igual que la línea anterior
this.Intercambiar(ref a, ref b);

// error de compilación
this.DrawLine(miLápiz, 10, 10, 100, 50);
```

En el ejemplo anterior se detecta un error; el problema es que estamos pidiendo a C# que localice el método `DrawLine` dentro del objeto actual. De hecho, `DrawLine` existe fuera del programa, en la clase `Graphics`, y debe ser invocado de la siguiente forma:

```
Graphics papel;
papel = pictureBox1.CreateGraphics();
papel.DrawLine(miLápiz, 10, 10, 100, 50);
```

## ● Sobrecarga de métodos

Nuestro método `Intercambiar` es útil en tanto puede trabajar con argumentos que tengan cualquier nombre. La desventaja es que tales argumentos deben ser enteros. Recordemos este código:

```
private void Intercambiar(ref int a, ref int b)
{
    int aCopia;
    aCopia = a;
    a = b;
    b = aCopia;
}
```

¿Pero qué tal si quisiéramos intercambiar dos variables `double`? Escribiríamos de manera intencional otro método `Intercambiar` con código diferente:

```
private void Intercambiar2(ref double a, ref double b)
{
    double aSegura = a;
    double bSegura = b;
    a = bSegura;
    b = aSegura;
}
```

Sin embargo, sería conveniente utilizar el mismo nombre para ambos métodos, y de hecho podemos hacerlo. He aquí cómo codificaríamos las declaraciones de estos métodos:

```
private void Intercambiar(ref int a, ref int b)
{
    int aCopia;
    aCopia = a;
    a = b;
    b = aCopia;
}

private void Intercambiar(ref double a, ref double b)
{
    double aSegura = a;
    double bSegura = b;
    a = bSegura;
    b = aSegura;
}
```

Ahora podemos invocar estos métodos:

```
int c = 3;
int d = 4;
double g = 1.2;
double h = 4.5;
Intercambiar(ref c, ref d);
Intercambiar(ref g, ref h);
```

¿Cómo decide C# cuál método utilizar? Hay dos métodos llamados `Intercambiar`, por lo que C# busca además del nombre el número de parámetros y sus tipos. Si en nuestro ejemplo `a` y `b` hubieran sido declaradas como variables `double`, C# invocaría el método que intercambia valores `double`. El código que contienen los métodos puede ser diferente; es el número y tipo de sus parámetros lo que determina cuál método será invocado. Al número de parámetros y el tipo de los mismos se le conoce como la *firma* del método.

Si el método devuelve un resultado, el tipo de este resultado no participa en la sobrecarga; es decir, son los tipos de los parámetros del método los que deben ser distintos.

Lo que hemos hecho aquí se denomina *sobrecargar* un método. El método `Intercambiar` ha sido sobrecargado con varias posibilidades.

Por lo tanto, si usted escribe métodos que realizan tareas similares pero tienen distintos números y/o tipos de argumentos, sería conveniente que utilizara la sobrecarga y eligiera el mismo nombre en vez de inventar artificialmente uno distinto.

Hay cientos de ejemplos del uso de la sobrecarga en las bibliotecas de C#. Por ejemplo, existen cuatro versiones del método `DrawLine`.

## ● Transferencia de objetos a los métodos

En nuestros ejemplos nos hemos concentrado en cómo pasar números, pero a veces también es necesario transferir objetos más complicados, como lápices y cuadros de imagen. El siguiente ejemplo requiere que pasemos dos números que se van a sumar, y también que transfiramos la etiqueta en donde se mostrará el resultado:

```
private void MostrarSuma(Label muestraResultado, int a, int b)
{
    muestraResultado.Text = Convert.ToString(a + b);
}
```

Ésta es la forma en que podríamos invocar el método:

```
MostrarSuma(label1, 3, 4);
MostrarSuma(label2, 5, 456);
```

Al pasar un objeto por valor podemos manipular sus propiedades e invocar sus métodos.

## Fundamentos de programación

- Un método es una sección de código que tiene asignado un nombre. Para invocar el método usamos su nombre.
- Podemos codificar métodos `void`, o métodos que devuelvan un solo resultado.
- Es posible pasar argumentos a un método. Esto puede hacerse por valor o por referencia (mediante el uso de `ref` o de `out`).
- Cuando un método sólo debe devolver un valor, la transferencia puede llevarse a cabo mediante el uso de `return` en vez de utilizar `out` o `ref`.
- Si podemos identificar una tarea bien definida en nuestro código, seremos capaces de separarla y escribirla como un método.

## Errores comunes de programación

- El encabezado del método debe incluir el nombre de los tipos. El siguiente código está mal debido precisamente a que no cumple esa condición:

```
private void MétodoUno(x)    // incorrecto
```

En su lugar debemos utilizar algo como esto:

```
private void MétodoUno(int x)
```

- La llamada a un método no debe incluir los nombres de los tipos. Por ejemplo, en vez de:

```
MétodoUno(int y);
```

debemos usar:

```
MétodoUno(y);
```

- Al invocar un método debemos suministrar el número y el tipo correctos de argumentos que lo componen.
- Siempre debemos “utilizar” de alguna forma los valores devueltos. El siguiente estilo de llamada incumple esta condición:

```
UnMétodo(e, f);
```

- Cuando un método especifica que requiere parámetros `out` o `ref`, la llamada debe confirmarlo, como en el siguiente ejemplo:

```
HacerTarea(ref x, out y);           // correcto
HacerTarea(x, y);                  // incorrecto

// el método:
private void HacerTarea(ref int a, out int b)
{
    ... cuerpo
}
```

## Secretos de codificación

- El patrón general de los métodos toma dos formas. En primer lugar, la manera de declarar un método que no devuelve un resultado es:

```
private void NombreMétodo(lista de parámetros)
{
    ... cuerpo
}
```

Este tipo de método debe ser invocado mediante una instrucción, como en el siguiente ejemplo:

```
NombreMétodo(lista de argumentos);
```

- En el caso de los métodos que devuelven un resultado, la declaración se hace de esta manera:

```
private tipo NombreMétodo(lista de parámetros)
{
    ... cuerpo
}
```

Además, puede especificarse cualquier tipo o clase para el valor devuelto.

- Es posible invocar el método como parte de una expresión, como en el siguiente ejemplo:

```
n = NombreMétodo(a, b);
```

- El cuerpo del método debe incluir una instrucción `return` con el tipo de valor correcto.

- Cuando un método no tiene argumentos debemos usar paréntesis vacíos ( ) tanto en la declaración como en la llamada.
- El escritor del método crea la lista de parámetros. Cada parámetro debe tener especificados su nombre, su tipo y la palabra clave `ref` o `out` (cuando su transferencia se hace por referencia).
- Quien hace la llamada al método escribe la lista de argumentos. Esta lista consiste de una serie de elementos escritos en el orden correcto (que coincida con los parámetros) y con los tipos correctos. A diferencia de los parámetros que forman parte de un método, en este caso no se utilizan los nombres de los tipos. Debemos emplear `out` o `ref` si el método lo estipula.

## Nuevos elementos del lenguaje

- La declaración de métodos privados.
- La invocación o llamada a un método, que consiste del nombre del método y sus argumentos.
- El uso de `return` para salir de un método que no sea `void` y devolver al mismo tiempo un valor.
- El uso de `return` para salir de un método `void`.
- El uso de `out` y `ref`.
- El uso de la sobrecarga.
- El uso de `this` para representar al objeto actual.

## Nuevas características del IDE

En este capítulo no se presentaron nuevas características del IDE.

## Resumen

- Los métodos contienen subtarefas de un programa.
- Podemos pasar (o transferir) argumentos a los métodos.
- La utilización de métodos se conoce como *invocarlos*.
- Los métodos que no son `void` devuelven un resultado.

## EJERCICIOS

Para probar los métodos que escribirá a continuación cree una GUI simple con cuadros de texto, etiquetas y cuadros de mensaje, según se requiera. Utilice un clic de botón para ejecutar su código.

El primer grupo de problemas sólo requiere métodos `void` y la transferencia de argumentos por valor:

- 5.1** Escriba un método llamado `MostrarNombre` con un parámetro `string`. Al ejecutar el programa el nombre suministrado debe desplegarse en un cuadro de mensaje.

- 5.2** Escriba un método llamado `MostrarNombres` con dos parámetros `string` que representen su primer nombre y su apellido paterno. El método debe mostrar su primer nombre en un cuadro de mensaje, y su apellido paterno en otro.
- 5.3** Escriba un método llamado `MostrarIngresos` con dos parámetros enteros que representen el salario de un empleado y el número de años trabajados. El método debe mostrar el total de ingresos obtenidos por el empleado en un cuadro de mensaje, suponiendo que haya obtenido la misma cantidad de ingresos cada año.
- 5.4** Codifique un método que dibuje un círculo, dadas las coordenadas de su centro y su radio. Su encabezado deberá ser el siguiente:

```
private void Círculo(  
    Graphics áreaDibujo, Pen lápizAUsar,  
    int xCentro, int yCentro, int radio);
```

- 5.5** Codifique un método llamado `DibujarCalle` que trace una calle llena de casas, para lo cual debe utilizar el método `DibujarCasa` que empleamos en este capítulo. Para el propósito de este ejercicio la calle debe incluir cuatro casas, con 20 píxeles de espacio entre cada una de ellas. Los argumentos deben proporcionar la ubicación y el tamaño de la casa que se halla en el extremo izquierdo del formulario, y deben ser idénticos a los de `DibujarCasa`.
- 5.6** Codifique un método que se llame `DibujarCalleEnPerspectiva`, de manera que tenga los mismos argumentos que el método creado en el ejercicio 5.5. Sin embargo, cada casa debe ser 20% más pequeña que la que se encuentre a su izquierda.

Los programas siguientes requieren métodos que devuelvan un resultado. Utilice argumentos que se transfieran por valor:

- 5.7** Escriba un método que devuelva el equivalente en pulgadas de su argumento en centímetros. La siguiente es una llamada de ejemplo:

```
double pulgadas = EquivalentePulgadas(10.5);
```

Multiplique los centímetros por 0.394 para calcular las pulgadas.

- 5.8** Escriba un método que devuelva el volumen de un cubo, dada la longitud de uno de sus lados. La siguiente es una llamada de ejemplo:

```
double vol = VolumenCubo(1.2);
```

- 5.9** Escriba un método que devuelva el área de un círculo, dado su radio como un argumento. La siguiente es una llamada de ejemplo:

```
double a = ÁreaCírculo(1.25);
```

El área del círculo se obtiene con base en la fórmula `Math.PI * r * r`. Aunque podríamos utilizar el número 3.14, `Math.PI` nos proporciona un valor más exacto.

- 5.10** Escriba un método llamado `EnSegundos` que acepte tres enteros, los cuales representarán el tiempo en horas, minutos y segundos. El método debe devolver el tiempo total en segundos. La siguiente es una llamada de ejemplo:

```
int totalSegundos = EnSegundos(1, 1, 2);    // devuelve 3662
```

- 5.11** Escriba un método que devuelva el área de un cilindro sólido. Decida cuáles parámetros utilizará. Su codificación deberá invocar el método `Áreacírculo` del ejercicio 5.9 para que le ayude a calcular el área de las partes superior e inferior (la circunferencia del círculo se obtiene mediante la fórmula  $2 * \text{Math.PI} * r$ .)

- 5.12** Escriba un método llamado `Incremento`, que sume 1 a su argumento entero. La siguiente es una llamada de ejemplo:

```
int n = 3;
int a = Incremento(n);    // devuelve 4
```

Los problemas que se plantean a continuación requieren el uso de parámetros por valor y/o parámetros por referencia, junto con métodos que pueden (o no) devolver valores:

- 5.13** Escriba un método llamado `SumaYDiferencia`, que calcule la suma y la diferencia de dos valores enteros cualesquiera (por ejemplo, si los argumentos de entrada son 3 y `n`, debe devolver `3+n` y `3-n`).
- 5.14** Escriba un método llamado `SegundosAHMS`, que reciba un número de segundos y los convierta en horas, minutos y segundos. Utilice los operadores `%` y `/` (por ejemplo, 3662 segundos equivalen a 1 hora, 1 minuto y 2 segundos).
- 5.15** Escriba un método llamado `DiferenciaTiempoEnSegs`, que tenga seis argumentos y devuelva un resultado entero. Debe recibir dos cantidades de tiempo en horas, minutos y segundos, y devolver en segundos la diferencia entre ellos. Para resolver este problema debe invocar el método `EnSegundos` (ejercicio 5.10) desde su método `DiferenciaTiempoEnSegs`.
- 5.16** Escriba un método llamado `HMSTranscurridos`, que acepte dos cantidades de tiempo en segundos y devuelva las horas, minutos y segundos transcurridos entre ellas. Su método debe utilizar el método `segundosAHMS` del ejercicio 5.14.
- 5.17** Escriba un método `void` llamado `Incremento`, que incremente su argumento entero. La siguiente es una llamada de ejemplo:

```
int v = 4;
Incremento(ref v);    // ahora v vale 5
```

Los siguientes problemas son acerca de la recarga de métodos:

- 5.18** Utilice cualquier programa que contenga el método `EnSegundos`. Agregue un método que también se llame `EnSegundos` y que tenga dos argumentos, uno para los minutos y otro para los segundos.
- 5.19** Utilice su programa del ejercicio 5.17, en donde se emplea el método `Incremento`. Escriba otras dos versiones de `Incremento`: una con un argumento `double` y otra con un argumento `string`. Este último método debe utilizar el operador `+` para unir un espacio al final de la cadena.

**SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**

**5.1** En (10, 20), (30, 10), (27, 26).

**5.2** En la primera llamada no se debe utilizar las comillas, ya que indican una cadena y no un entero.

En la segunda llamada el papel y el lápiz están en orden incorrecto.

En la tercera llamada falta un argumento.

**5.3** Aparecerá un cuadro de mensaje mostrando el texto `Naranjas`.

**5.4** El cuadro de mensaje muestra el valor original de `a`, que es 3. La `a` que se convierte en 11 dentro del método es una variable local.

**5.5** Éstas son las etapas para sustituir una llamada por su resultado. Para:

```
n = ÁreaRectángulo(10, 20);
```

tenemos:

```
n = 200;
```

Para la línea:

```
MessageBox.Show("el área mide " +  
    Convert.ToString(ÁreaRectángulo(3, 4)));
```

tenemos las siguientes etapas:

```
MessageBox.Show("el área mide " + Convert.ToString(12));  
MessageBox.Show("el área mide 12");
```

Para la línea:

```
n = ÁreaRectángulo(10, 20) * ÁreaRectángulo(7,8);
```

tenemos las etapas:

```
n = 200 * 56;  
n = 11200;
```

**5.6** Los valores que recibe `r` son:

```
6  
8  
7  
18  
12  
16  
14  
24
```

**5.7** No. Varias variables podrían contener el mismo valor, por lo que éstos no son únicos. Para utilizar una analogía: cada casa tiene una dirección y un valor (por ejemplo, el número de personas que viven en ella). Sería imposible rastrear la dirección de una casa si sólo contáramos con la información de que está habitada por tres personas, ya que probablemente hay muchas con esa característica.



**5.8** En principio, el método es idéntico al del ejemplo Método dólares. El código es:

```
private void CalcularBilletes(int cantidadUsuario,
    out int diez, out int uno)
{
    diez = cantidadUsuario / 10;
    uno = cantidadUsuario % 10;
}
```

**5.9** Funcionaría de manera correcta, aunque los nombres no son muy ilustrativos. El escritor de un método tiene la libertad de elegir los nombres de sus parámetros, y no es necesario que éstos coincidan con los nombres utilizados por quien invoca el método. Sin embargo, es responsabilidad del que hace la llamada proveer los argumentos en el orden correcto de izquierda a derecha.

**5.10** El valor de `x` queda como 4, y el de `y` cambia a 10. El método tiene un parámetro por valor y un parámetro `ref`. Dentro del método, cambiar la `a` a 5 sólo tiene un efecto local. La acción de pasar por valor evita que la variable `x` original se modifique.

**5.11** En la primera llamada se omitió la palabra clave `ref`. Esto produce un error de compilación. En la segunda llamada proveemos un número en vez de una variable. El número es una cantidad fija, y no una ubicación de memoria cuyo contenido se puede modificar. Por lo tanto, también se produce un error de compilación.



# Uso de los objetos

## En este capítulo conoceremos:

- las variables de instancia y `private`;
- el constructor de formularios;
- cómo utilizar las clases de la biblioteca;
- cómo utilizar `new`;
- cómo utilizar los métodos y las propiedades;
- la clase `Random`;
- las clases `TrackBar` y `Timer`.

## ● Introducción

---

A lo largo de este capítulo hablaremos con más detalle de los objetos. Analizaremos especialmente el uso de los distintos tipos de objetos que conforman la biblioteca de clases de C#. Tenga en cuenta que, aunque hay cientos ellos, los principios para usarlos son similares en todos los casos.

Ésta es una analogía: para leer un libro (cualquiera que sea) hay que abrirlo por su parte frontal, leer una página y avanzar a la siguiente; sabemos qué hacer con el libro. Lo mismo pasa con los objetos: después de usar unos cuantos sabemos qué esperar cuando se nos presenta uno nuevo.

En general los objetos que utilizaremos se denominan controles o componentes. Estos términos son casi sinónimos, aun cuando C# utiliza el término “control” para referirse a los elementos que pueden ser manipulados en un formulario (como los botones).

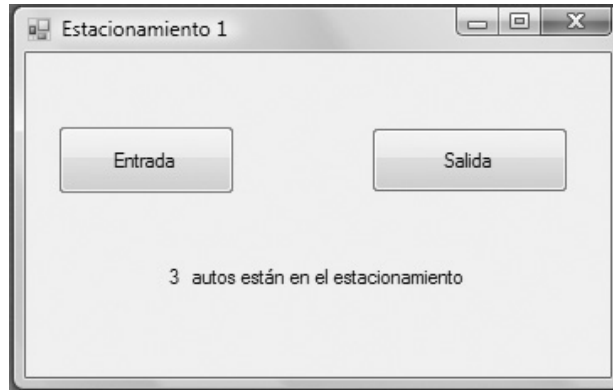


Figura 6.1 El programa Estacionamiento 1.

## ● Variables de instancia

Para poder enfrentar problemas más avanzados que los que hemos comentado hasta este punto, es necesario que conozcamos un nuevo lugar en dónde declarar variables. Hemos utilizado ya las palabras clave (o reservadas) `int`, `string` y otras para declarar variables locales dentro de los métodos, pero éstas son incapaces de lidiar por sí solas con todos los problemas.

A continuación veremos un programa simple (Estacionamiento 1, cuya GUI se ilustra en la Figura 6.1) para ayudar a operar un estacionamiento. Tiene dos botones: “entrada” y “salida”. El empleado hace clic en el botón apropiado a medida que un automóvil ingresa o se retira. El programa lleva la cuenta del número de automóviles que hay en el estacionamiento, y despliega el dato en una etiqueta.

Observe que el contador se modifica en dos métodos, así que no se puede declarar como variable local dentro de un solo método. Es tentador pensar en esta posibilidad, pero si la variable se declara dentro de *cada* método tendríamos dos variables separadas. He aquí el código:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace Estacionamiento_1
{
    public partial class Form1 : Form
    {
        private int cuentaAutos = 0;

        // aquí omitimos código no relevante para este ejemplo
    }
}
```

```

        private void botónEntrada_Click(object sender, EventArgs e)
        {
            cuentaAutos = cuentaAutos + 1;
            etiquetaConteo.Text = Convert.ToString(cuentaAutos);
        }

        private void botónSalida_Click(object sender, EventArgs e)
        {
            cuentaAutos = cuentaAutos - 1;
            etiquetaConteo.Text = Convert.ToString(cuentaAutos);
        }
    }
}

```

C# creó de manera automática una clase llamada `Form1`. Hemos agregado dos métodos a la clase: `botónEntrada_Click` y `botónSalida_Click` (cambiamos el nombre de los botones para que el código sea más comprensible). Además, establecimos en tiempo de diseño la propiedad `Text` de `etiquetaConteo` como 0.

Lo importante en este caso es la declaración de la variable `cuentaAutos`, pero antes de analizarla necesitamos repasar todo el código e identificar las distintas secciones que lo constituyen. Como podrá ver en su pantalla, el IDE ha generado una gran cantidad de código, y debemos conservar la mayor parte del mismo como esté. Sin embargo, hay momentos en los que necesitamos insertar nuestras propias instrucciones en medio del código creado por el IDE. La estructura, similar en prácticamente todos los programas, es la siguiente:

- Localice los elementos `using` en la parte superior del código. Estos elementos se utilizan para incorporar a nuestro programa grupos de bibliotecas de clases previamente escritas. Algunas veces necesitamos agregar nuestros propios elementos `using` en esta sección, como veremos más adelante en el capítulo. Los listados que se presentan en este libro no siempre muestran los elementos `using`, debido a que son idénticos en casi todos los casos.
- Localice el elemento `namespace`. Ésta es una característica avanzada que no explicaremos aquí. En nuestros listados de código impresos omitiremos la instrucción `namespace` junto con las llaves de apertura y de cierre que la acompañan. Esto nos deja el siguiente código:

```

public partial class Form1 : Form
{
    private int cuentaAutos = 0;

    // aquí omitimos código no relevante para este ejemplo

    private void botónEntrada_Click(object sender, EventArgs e)
    {
        cuentaAutos = cuentaAutos + 1;
        etiquetaConteo.Text = Convert.ToString(cuentaAutos);
    }

    private void botónSalida_Click(object sender, EventArgs e)
    {
        cuentaAutos = cuentaAutos - 1;
        etiquetaConteo.Text = Convert.ToString(cuentaAutos);
    }
}

```

Aunque no siempre mostraremos los elementos `using` y la instrucción `namespace`, usted siempre deberá tenerlos en sus programas. ¡No los elimine!

- Por último llegamos a la clase, que contiene varias variables y métodos privados y públicos. Identifique la línea `public partial class` y su correspondiente llave de apertura (`{`). Debajo de estas líneas podemos colocar las declaraciones de nuestras variables `private`.

Una vez que hemos visto el patrón general del código creado por el IDE de C#, nos enfocaremos en la variable `cuentaAutos`:

- Esta variable se declara *fuera* de los métodos, pero *dentro* de la clase `Form1`; en consecuencia, cualquier método que se halle en `Form1` puede utilizarla.
- Se ha declarado como `private`, lo cual significa que cualquier otra clase que pudiéramos llegar a tener no podrá emplearla. La variable está *encapsulada* o sellada dentro de `Form1`; es decir, su propósito es que la utilicen los métodos y propiedades de `Form1` solamente.
- `cuentaAutos` es un ejemplo de una *variable de instancia*. Pertenecce a una instancia de una clase, en vez de pertenecer a un método. Otro término para denominarla es variable “a nivel de clase”.
- Se dice que `cuentaAutos` tiene *alcance de clase*. El alcance de un elemento es el área del programa en donde puede ser utilizada. El otro tipo de alcance que hemos visto es el local, que se emplea con las variables locales.
- Por lo general las variables de instancia se declaran como `private`.
- Por convención, en C# no se pone en mayúscula la primera letra de las variables de instancia.

Hay que considerar que el programador tiene la libertad de elegir los nombres para las variables de instancia. ¿Pero qué pasa si uno de ellos coincide con el nombre de una variable local, como en el siguiente ejemplo?

```
public partial class Form1 : Form
{
    private int n = 8;

    private void MiMétodo()
    {
        int n;
        n = 3;    //¿cuál n?
    }
}
```

Aunque ambas variables son accesibles (en alcance) dentro de `MiMétodo`, de acuerdo con la regla el programa elegirá la variable local. Por lo tanto, la variable de instancia (nivel de clase) `n` permanecerá en 8.

## PRÁCTICA DE AUTOEVALUACIÓN

**6.1** ¿Cuáles serían las consecuencias de eliminar la declaración local de `n` en la clase `Form1` anterior?

Las variables de instancia son esenciales, pero no debemos dejar de lado las variables locales. Por ejemplo, si una variable sólo se utiliza dentro de un método y no necesitamos mantener su valor entre una llamada y otra, es mejor hacerla local.

## ● El constructor del formulario

---

Regresemos al programa del estacionamiento. Utilizamos una variable llamada `cuentaAutos` para contar, y una etiqueta para mostrar el valor de dicha variable. Establecimos el valor de `cuentaAutos` en 0 dentro del programa, y dimos el valor 0 a la propiedad de texto de `etiquetaConteo` en tiempo de diseño. De hecho estos valores no son independientes. Considere la posibilidad de que haya cinco automóviles en el estacionamiento por un tiempo largo. En ese caso tendríamos que alterar tanto el valor inicial de `cuentaAutos` como el de la propiedad de texto de `etiquetaConteo`. En realidad sólo hay un elemento que contiene el número de automóviles, y es `cuentaAutos`. En vez de establecer por separado el valor de texto inicial de `etiquetaConteo` en tiempo de diseño, sería mejor colocar el valor de `cuentaAutos` (cualquiera que éste sea) en la etiqueta para que se despliegue cuando el programa inicie su ejecución.

Es común que los valores iniciales de los controles dependan de variables y de otros controles. Podríamos tratar de establecer esta situación en tiempo de diseño, pero cuando hay varios controles aumenta la posibilidad de cometer errores, además de que no se expresan las dependencias. Es mejor si establecemos valores iniciales relacionados en el código. Por fortuna, C# cuenta con un área especial del programa para este tipo de inicializaciones que se harán una sola vez. He aquí la segunda versión del programa, llamada Estacionamiento 2.

```
public partial class Form1 : Form
{
    private int cuentaAutos = 0;

    public Form1()
    {
        InitializeComponent();
        etiquetaConteo.Text = Convert.ToString(cuentaAutos);
    }

    private void botónEntrada_Click(object sender, EventArgs e)
    {
        cuentaAutos = cuentaAutos + 1;
        etiquetaConteo.Text = Convert.ToString(cuentaAutos);
    }
}
```

```
private void botónSalida_Click(object sender, EventArgs e)
{
    cuentaAutos = cuentaAutos - 1;
    etiquetaConteo.Text = Convert.ToString(cuentaAutos);
}
}
```

Recuerde que por lo general omitimos las líneas `using` que están en la parte superior del código, y también la línea `namespace` con sus llaves de apertura `{`, y de cierre `}`.

Localice el encabezado:

```
public Form1()
{
```

Esto inicia una sección de código que es una especie de método, aunque no incluye la especificación `void` ni establece un tipo de valor de retorno. Su nombre (`Form1`) coincide con el de la clase, y el método se conoce como *constructor* de la clase `Form1`.

Cuando el sistema de C# ejecuta su programa, lo primero que hace es invocar al constructor del formulario. La función del constructor consiste en crear el formulario e inicializarlo (el constructor se declara como `public` en vez de `private` debido a que es invocado desde el exterior de la clase).

Si analiza el código creado para el constructor, se dará cuenta de que contiene sólo la siguiente invocación a un método:

```
InitializeComponent();
```

cuya función es colocar componentes en el formulario. Después de esta línea podemos insertar nuestras propias instrucciones para realizar más operaciones de inicialización. En el caso de este programa colocamos la siguiente línea:

```
etiquetaConteo.Text = Convert.ToString(cuentaAutos);
```

En esta versión mejorada del programa no tenemos necesidad de establecer la propiedad `Text` de la etiqueta en tiempo de diseño; en vez de ello insertamos código para realizar esta tarea en el constructor, para garantizar que la propiedad `Text` tenga el mismo valor que la variable `cuentaAutos`.

**PRÁCTICA DE AUTOEVALUACIÓN**

---

**6.2** ¿Cuál es el error en el siguiente código?

```
Public Form1()  
{  
    ... etc  
  
    label1.Text = "38";  
    InitializeComponent();  
}
```

En el ejemplo anterior modificamos el constructor, pero no lo invocamos nosotros. Más adelante en este capítulo crearemos nuevos objetos al invocar de manera explícita el constructor de su clase.

---

**● La clase `TrackBar`**

---

Veamos otro ejemplo de inicialización de componentes. La `TrackBar` o barra de seguimiento es un componente de la GUI que está disponible en el cuadro de herramientas. En principio es similar a la barra de desplazamiento que se encuentra en la parte lateral de la ventana de un procesador de textos, sólo que éste puede colocarse en cualquier parte de un formulario, permitiendo que el usuario arrastre su control de posición (o control deslizante) al lugar requerido; sus valores máximo y mínimo pueden establecerse mediante propiedades, tanto en tiempo de diseño como en tiempo de ejecución.

La barra de seguimiento no se utiliza para manipular valores precisos como edades, cuyo rango podría ser tan extenso como de 10 a 100. Más bien se usa para establecer valores más informales, como el volumen de un altavoz.

A continuación veremos un programa (Forma de óvalo) que permite al usuario modificar el ancho y la altura de una ellipse. Las dimensiones actuales se despliegan mediante etiquetas en el formulario. La Figura 6.2 muestra la interfaz resultante del código siguiente:

```
public partial class Form1 : Form  
{  
    private Graphics papel;  
  
    public Form1()  
    {  
        InitializeComponent();  
        papel = pictureBox1.CreateGraphics();  
        trackBarVertical.Minimum = 0;  
        trackBarVertical.Maximum = pictureBox1.Height;  
        etiquetaVertical.Text = Convert.ToString(trackBarVertical.Value);  
    }  
}
```





Figura 6.2 El programa Forma de óvalo.

```
trackBarHorizontal.Minimum = 0;
trackBarHorizontal.Maximum = pictureBox1.Width;
etiquetaHorizontal.Text = Convert.ToString(trackBarHorizontal.Value);
}

private void trackBarVertical_Scroll(object sender,
EventArgs e)
{
    SolidBrush miPincel = new SolidBrush(Color.Black);
    etiquetaVertical.Text = Convert.ToString(trackBarVertical.Value);
    papel.Clear(Color.White);
    papel.FillEllipse(miPincel, 0, 0, trackBarHorizontal.Value,
        trackBarVertical.Value);
}

private void trackBarHorizontal_Scroll(object sender,
EventArgs e)
```

```

    {
        SolidBrush miPincel = new SolidBrush(Color.Black);
        etiquetaHorizontal.Text = Convert.ToString(trackBarHorizontal.Value);
        papel.Clear(Color.White);
        papel.FillEllipse(miPincel, 0, 0, trackBarHorizontal.Value,
            trackBarVertical.Value);
    }
}

```

Veamos ahora algunos puntos sobre la inicialización en tiempo de diseño:

- Cambiamos el nombre de las barras de seguimiento a `trackBarHorizontal` y `trackBarVertical`.
- Para colocar en vertical una barra de seguimiento hay que establecer su propiedad **Orientation** en **Vertical**.
- Modificamos el nombre de las etiquetas a `etiquetaHorizontal` y `etiquetaVertical`.
- Establecimos el tamaño (es decir, su propiedad `Size`) del cuadro de imagen en 150, 150.

Utilizamos el constructor en tiempo de ejecución para inicializar algunos componentes:

- Establecimos la propiedad **Minimum** de las barras de seguimiento en 0, y las propiedades **Maximum** en la altura y el ancho del cuadro de imagen.
- El valor inicial de la propiedad **Text** de `etiquetaHorizontal` —que muestra el valor actual de la barra de seguimiento— se establece en `trackBarHorizontal.Value`; `trackBarVertical` se inicializa en manera similar.

Tenga en cuenta que:

- El método del evento de la barra de seguimiento (`Scroll`) se invoca cuando la desplazamos a una nueva posición.
- La propiedad **value** de la barra de seguimiento nos proporciona el valor actual. Utilizamos esta propiedad para controlar el tamaño de un rectángulo imaginario que encierra al óvalo.
- El área de dibujo es utilizada por dos métodos, por lo cual debe declararse como una variable de instancia a nivel de clase, antes de los métodos.

Este programa ilustra los beneficios de inicializar componentes en el constructor del formulario.

#### PRÁCTICA DE AUTOEVALUACIÓN

**6.3** En el ejemplo de la barra de seguimiento, ¿cuáles son las consecuencias de alterar el tamaño de la barra de seguimiento en tiempo de diseño?

## ● La palabra clave `using` y los espacios de nombres

C# incluye una enorme biblioteca (o colección) de clases que podemos utilizar. Un aspecto muy importante de la programación en C# radica en la posibilidad de emplear estas clases en vez de escribir nuestro propio código. A esto se le conoce como “reutilización de software”.

Debido a que hay miles de ellas, las clases están subdivididas en grupos cuyo ámbito es conocido mediante la palabra clave `namespace` o *espacio de nombre*. Por otro lado, para utilizar una clase primero debemos asegurarnos de que se *importe* a nuestro programa mediante la palabra clave `using`. Esto puede ocurrir de dos formas: algunos de los espacios de nombres que se utilizan con más frecuencia se importan de manera automática en cualquier aplicación de Windows. Estos espacios de nombres son:

```
System
System.Drawing
System.Collections
System.ComponentModel
System.Windows.Forms
System.Data
```

En este caso es preciso decidir:

- si la clase que requerimos se encuentra en uno de los espacios de nombres anteriores; de ser así, podemos utilizarla sin requerir acción adicional alguna;
- si la clase que requerimos no se encuentra en uno de los espacios de nombres anteriores; en tal situación tendremos que emplear una instrucción `using` en la parte superior de nuestro programa.

Veamos un ejemplo. Cuando utilicemos los archivos en el capítulo 18 aprenderemos a usar la clase `StreamReader`, como en el siguiente código:

```
// declarar un objeto StreamReader, llamado miFlujo
StreamReader miFlujo;
```

La clase `StreamReader` se encuentra en el espacio de nombres `System.IO`, por lo que debemos colocar la línea:

```
using System.IO;
```

en la parte superior de nuestro código.

Respecto de este tema, es preciso tener en cuenta dos consideraciones:

- Las instrucciones `using` no funcionan de manera jerárquica. Al importar el espacio de nombres `System` no se importan de manera automática todos los espacios de nombres que empiecen con `System`. Cada espacio de nombres debe importarse explícitamente.
- La instrucción `using` sólo es un método abreviado. Por ejemplo, podríamos utilizar la clase `StreamReader` sin necesidad de importarla, pero en ese caso tendríamos que incluir la siguiente línea:

```
System.IO.StreamReader miFlujo;
```

En resumen, la vasta biblioteca de clases de C# está organizada en espacios de nombres, los cuales podemos importar a cualquier programa. Una vez que importemos la clase tendremos que saber cómo crear una nueva instancia y cómo utilizar sus propiedades y métodos. Analizaremos todos estos aspectos mediante diversos ejemplos.

## ● Miembros, métodos y propiedades

---

Los *miembros* de una clase son sus propiedades y sus métodos. Las propiedades contienen los valores que representan el estado actual de una instancia de una clase (como el texto que contiene una etiqueta), mientras que los métodos hacen que la instancia realice una tarea; por ejemplo, dibujar un círculo.

Podemos utilizar las propiedades de manera similar a como lo hacemos con las variables: colocando un nuevo valor en ellas y accediendo a su valor actual. Como ejemplo veamos la forma en que podrían manipularse las propiedades `Width` y `Height` de una etiqueta:

```
// establecer un nuevo valor en una propiedad:
label1.Height = 30;
label1.Height = Convert.ToString(textBox1.Text);

// obtener el valor actual de la propiedad:
int a;
a = label1.Height;
a = label1.Height * label1.Width;
```

De acuerdo con la terminología de C#, es posible *establecer* una propiedad a un nuevo valor y *obtener* el valor actual de una propiedad. Cada una de estas propiedades tiene también un tipo. Por ejemplo, la propiedad `Width` de la etiqueta contiene un entero, mientras que la propiedad `Text` contiene una cadena de caracteres. Los nombres y tipos de las propiedades están disponibles en el sistema de Ayuda.

### PRÁCTICAS DE AUTOEVALUACIÓN

---

**6.4** Imagine un reproductor de CD; liste algunos métodos y propiedades que pueda tener. ¿Cuáles de estos métodos y propiedades podrían ser miembros?

**6.5** ¿Qué realizan las siguientes instrucciones en términos geométricos?

```
int a;
a = label1.Width * label1.Height;
label1.Height = label1.Width;
```

## ● La clase `Random`

---

En esta sección hablaremos sobre una clase (`Random`) cuyas instancias debemos declarar e inicializar de manera explícita. Los números aleatorios son muy útiles en simulaciones y juegos; por ejemplo, podemos proporcionar al jugador una situación inicial distinta cada vez que juegue. Las instancias de la clase `Random` nos proporcionan un “flujo continuo” de números que podemos obtener uno a la vez mediante el método `Next`. A continuación veremos un programa (Adivinador) que intenta adivinar la edad del usuario (de manera muy ineficiente), para lo cual muestra una secuencia de números aleatorios. Cuando usted hace clic en “correcto” el programa despliega el número de intentos que requirió para adivinar la edad. En la Figura 6.3 se muestra la interfaz del programa; el código es el siguiente:

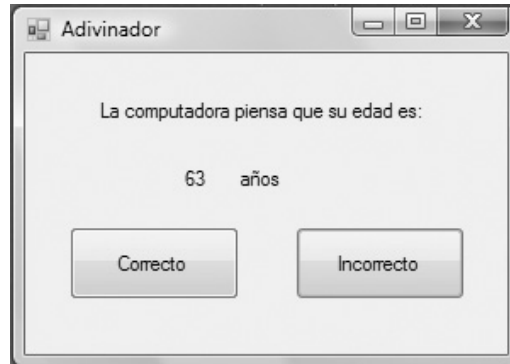


Figura 6.3 El programa Adivinador.

```
public partial class Form1 : Form
{
    private Random adivinadorEdad = new Random();
    private int intentos = 0;

    public Form1()
    {
        InitializeComponent();
        etiquetaAdivina.Text =
            Convert.ToString(adivinadorEdad.Next(5, 110));
    }

    private void botónCorrecto_Click(object sender,
        EventArgs e)
    {
        intentos = intentos + 1;
        MessageBox.Show("Número de intentos: " +
            Convert.ToString(intentos));
        intentos = 0;
        etiquetaAdivina.Text =
            Convert.ToString(adivinadorEdad.Next(5, 110));
    }
}
```

```
private void botónIncorrecto_Click(object sender,
    EventArgs e)
{
    etiquetaAdivina.Text =
        Convert.ToString(adivinatorEdad.Next(5, 110));
    intentos = intentos + 1;
}
}
```

Para utilizar una nueva clase debemos localizar en el sistema de Ayuda su espacio de nombres y colocar la instrucción `using` apropiada. La clase `Random` se halla en el espacio de nombres `System`, el cual se importa de manera automática. No se requieren instrucciones `using` adicionales.

Ahora necesitamos declarar e inicializar una instancia de nuestra clase. Podemos hacerlo de dos formas. La primera es utilizar una instrucción como la siguiente:

```
private Random adivinatorEdad = new Random();
```

Observe que:

- Elegimos el nombre `adivinatorEdad` para nuestra instancia.
- La instrucción invoca al constructor de la clase `Random`, el cual siempre tiene el mismo nombre que la clase en sí. Básicamente el constructor es un método.
- La palabra `new` se antepone al uso del constructor. La instrucción `new` crea una nueva instancia de una clase en la RAM.
- Los constructores pueden sobrecargarse, por lo que deberá elegir el más conveniente. `Random` tiene dos constructores y, en este caso, el adecuado es el que no tiene parámetros.
- Podemos considerar que la instrucción consta de dos partes:

```
private Random adivinatorEdad...
```

y:

```
... = new Random();
```

La primera parte declara `adivinatorEdad` como una variable de la clase `Random`, pero no tiene todavía una instancia concreta (que contenga métodos y valores de propiedades) asociada a ella. La segunda parte invoca al constructor de la clase `Random` para completar las tareas de declaración e inicialización.

La segunda forma de declarar e inicializar instancias es mediante instrucciones de declaración e inicialización en distintas áreas del programa, como en el siguiente ejemplo:

```
public partial class Form1 : Form
{
    private Random adivinatorEdad;
    ...

    adivinatorEdad = new Random();
}
```

Sin importar el método que seleccionemos, es necesario que tengamos en cuenta varios puntos:

- La declaración establece la clase de la instancia. En este caso es una instancia de `Random`.
- La declaración establece el alcance del objeto. En este caso, `adivinatorEdad` tiene alcance de clase y, por lo tanto, puede utilizarse en cualquier método de la clase `Form1` en vez de ser local para un método.
- `adivinatorEdad` es un objeto privado (`private`). No se puede emplear en otras clases que no sean nuestra clase `Form1`. Por lo general damos la categoría de privadas a dichas variables.
- La inicialización debe estar dentro del constructor del formulario o dentro de otro método.
- Siempre que pueda utilizar la forma de una sola línea de declaración e inicialización, hágalo.

¿Por qué sería necesario separar la declaración y la inicialización? Es usual que se presente la necesidad de contar con una variable de instancia (y no una variable local), misma que debe declararse fuera de los métodos. Pero en ocasiones el objeto sólo puede ser inicializado cuando el programa empieza a ejecutarse, tal vez mediante la introducción de un valor por parte del usuario, dato que deberá pasarse como parámetro al constructor.

En este caso colocaríamos el código de inicialización dentro de un método (o tal vez en el constructor del formulario). No podemos colocar la declaración dentro del método, pues el elemento se declararía como local.

Sigamos analizando el programa en donde utilizamos el objeto `Random`. Hemos creado una instancia de la clase `Random` llamada `adivinatorEdad`, pero aún nos faltan los números aleatorios reales.

Tan pronto como creamos un objeto con `new` estamos en disposición de emplear sus propiedades y métodos. La documentación nos indica que varios métodos nos proporcionan un número aleatorio, y para este caso elegimos aquel que nos permita especificar el rango de los números. Este método se llama `Next` (debido a que obtiene el siguiente número aleatorio de una secuencia de números). Así, colocamos en nuestro programa la siguiente instrucción:

```
etiquetaAdivina.Text =
    Convert.ToString(adivinatorEdad.Next(5, 110));
```

También podríamos haber codificado esta instrucción de manera menos concisa:

```
int adivina;
adivina = adivinatorEdad.Next(5, 110);
etiquetaAdivina.Text = Convert.ToString(adivina);
```

Se eligió el rango de números aleatorios de 5 a 110, ambos inclusive, para representar los límites de las edades. En resumen, declaramos una instancia de la clase apropiada (`Random`) y utilizamos `new` para crearla e inicializarla. Estas dos etapas pueden ser combinadas o separadas, dependiendo del programa específico en el que estemos trabajando. Después utilizamos las propiedades y métodos de la instancia. La documentación nos proporciona los detalles sobre sus nombres y los tipos de datos/parámetros requeridos.

**PRÁCTICA DE AUTOEVALUACIÓN**

**6.6** Fui a la agencia de automóviles, y después de ver uno de sus folletos ordené un Netster de 5 litros, fabricado a la medida, de color azul. Cuando llegó me puse a conducirlo. ¿La clase `Auto` tiene un constructor? ¿El constructor cuenta con parámetros? ¿Cuál es la instancia, la fotografía del automóvil o el automóvil real?

### ● La clase `Timer`

Las clases que hemos utilizado hasta el momento se encuentran en uno de dos grupos:

- Las del cuadro de herramientas, como los botones. Estas clases tienen una representación en tiempo de diseño en el formulario (por ejemplo, es posible cambiar su tamaño). Incluyen plantillas de código de manejo de eventos, y el código para invocar a sus constructores se genera de manera automática. Además, podemos establecer sus propiedades iniciales en tiempo de diseño.
- Las de las bibliotecas, que no cuentan con una representación visual (como `Random`) y no aparecen en tiempo de diseño. Otra de sus características es que tenemos que codificar de manera explícita las invocaciones a sus constructores, y sólo podemos establecer sus propiedades en tiempo de ejecución.

El temporizador (objeto de la clase `Timer`) es un poco distinto: se encuentra en el cuadro de herramientas, pero cuando se pone en un formulario el IDE abre una nueva ventana de la **Bandeja de componentes** y coloca un icono de temporizador (un reloj) en ella. Podemos establecer sus propiedades en tiempo de diseño, y al hacer doble clic en el icono aparece el código para manejar sus eventos. Al ejecutar el programa el temporizador no aparece en el formulario.

Las siguientes son las principales características del temporizador:

- Crea un mecanismo para ejecutar eventos recurrentes a intervalos regulares (tics). Cada intervalo es un evento que invoca el método `Tick`.
- La propiedad `Interval` puede establecerse como un valor entero que representa en milisegundos el tiempo entre tics.
- Nos permite iniciarlo y detenerlo con los métodos `Start` y `Stop`.
- Da la posibilidad de colocar cualquier número de temporizadores en un programa, cada uno con un intervalo distinto.

Veamos ahora un programa (Gotas de lluvia) que simula una hoja de papel bajo la lluvia. Este programa muestra cómo caen gotas de un tamaño aleatorio, a intervalos también aleatorios. Estos intervalos pueden modificarse mediante una barra de seguimiento. En la Figura 6.4 se muestra la interfaz del programa, cuyo código se incluye a continuación:

```
public partial class Form1 : Form
{
    private Random númeroAleatorio = new Random();
    private Graphics papel;

    public Form1()
    {
```



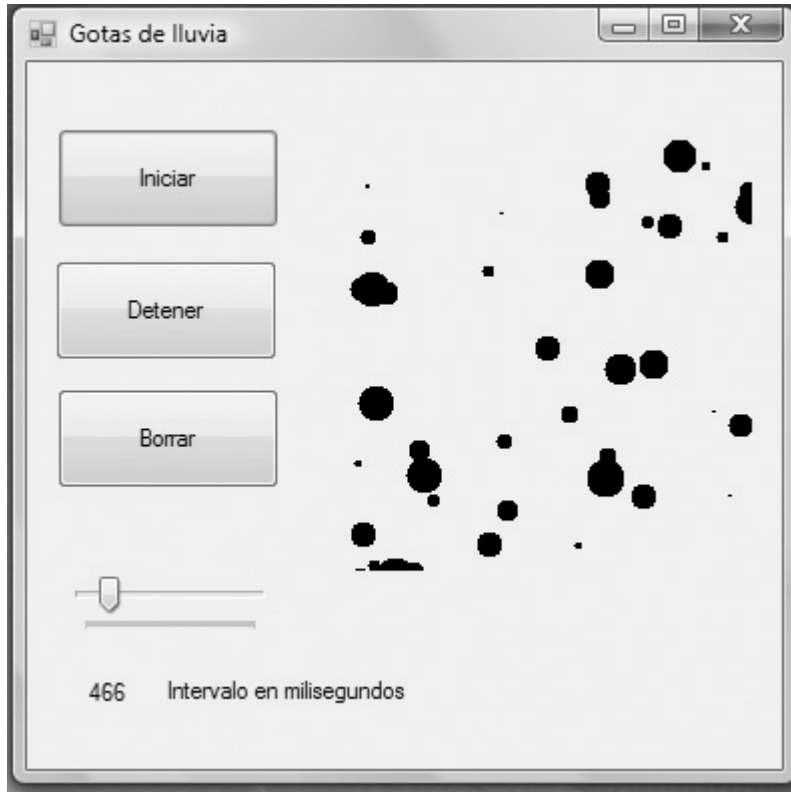


Figura 6.4 El programa Gotas de Lluvia.

```
InitializeComponent();  
papel = pictureBox1.CreateGraphics();  
etiquetaIntervalo.Text = Convert.ToString(trackBar1.Value);  
}  
  
private void botónIniciar_Click(object sender,  
    EventArgs e)  
{  
    timer1.Start();  
}  
  
private void botónDetener_Click(object sender,  
    EventArgs e)  
{  
    timer1.Stop();  
}
```

```

private void botónBorrar_Click(object sender,
    EventArgs e)
{
    papel.Clear(Color.White);
}

private void trackBar1_Scroll(object sender,
    EventArgs e)
{
    int intervaloTiempo = trackBar1.Value;
    etiquetaIntervalo.Text = Convert.ToString(intervaloTiempo);
}

private void timer1_Tick(object sender,
    EventArgs e)
{
    int x, y, tamaño;
    Brush miPincel = new SolidBrush(Color.Black);

    x = númeroAleatorio.Next(0, pictureBox1.Width);
    y = númeroAleatorio.Next(0, pictureBox1.Height);
    tamaño = númeroAleatorio.Next(1, 20);
    papel.FillEllipse(miPincel, x, y, tamaño, tamaño);

    // establece nuevo intervalo para el temporizador
    timer1.Stop();
    timer1.Interval =
        númeroAleatorio.Next(1, trackBar1.Value);
    timer1.Start();
}
}

```

En cada tic el programa dibuja un círculo relleno de tamaño y posición aleatorios. También restablecimos el intervalo de tiempo a un valor aleatorio controlado por la barra de seguimiento (para esto se requiere detener e iniciar el temporizador). Cada vez que se mueve la barra de seguimiento, su valor actual se despliega en una etiqueta. Elegimos los valores de las propiedades **Minimum** y **Maximum** de la barra de seguimiento mediante experimentación, concluyendo con 200 y 2000, respectivamente; establecimos estos valores en tiempo de diseño. También utilizamos el método **clear** del cuadro de imagen, el cual da a todo el cuadro un color especificado.

#### PRÁCTICA DE AUTOEVALUACIÓN

**6.7** Tenemos un temporizador con un intervalo de 1000, es decir, un segundo. Explique cómo podemos mostrar los minutos en el formulario.

## Fundamentos de programación

Durante mucho tiempo el sueño de los programadores ha sido poder construir programas de la misma forma en que se construyen los sistemas de alta fidelidad; es decir, a partir de componentes listos para usar, como altavoces, amplificadores, controles de volumen, etc. El surgimiento de la programación orientada a objetos, y las numerosas bibliotecas de clases como las que ofrece el marco de trabajo .NET, hacen que esto sea cada vez una realidad más cercana.

Además de poder utilizar los componentes existentes, también podemos aprovechar C# para escribir componentes de GUI, de manera que estén disponibles para otras personas. La incorporación de dichos componentes es simple: se agregan a un proyecto mediante la acción de un menú. De ahí en adelante aparecen en el cuadro de herramientas, igual que cualquier otro control, y proporcionan encabezados de métodos para manejo de eventos y propiedades que pueden establecerse en tiempo de diseño. En términos prácticos, sin embargo, vale la pena buscar un control existente que cumpla con nuestros requerimientos, en vez de tratar de reinventar la rueda codificando desde cero.

## Errores comunes de programación

Si se declara una instancia pero se omite su inicialización con `new`, se producirá un error en tiempo de ejecución del tipo `System.NullReferenceException`. Los errores en tiempo de ejecución (o *bugs*, como también se les conoce) resultan más problemáticos que los errores en tiempo de compilación, pues son más difíciles de encontrar y más graves, en tanto detienen la ejecución del programa. ¡Le garantizamos que tarde o temprano se enfrentará a errores de este tipo!

## Secretos de codificación

- Las variables de instancia se declaran fuera de los métodos, mediante el uso de la palabra clave `private`, como en el siguiente ejemplo:

```
private int suVariable;
private Random miVariable = new Random();
```

- Las variables de instancia pueden ser inicializadas al momento de declararla o dentro de un constructor o método.
- Podemos manipular las propiedades de manera similar a como lo hacemos con las variables; estableciendo y obteniendo sus valores.

## Nuevos elementos del lenguaje

- Variables de instancia privadas.
- Uso de `new` para la inicialización.
- Uso de la instrucción `using` para importar espacios de nombres.
- Las clases `TrackBar`, `Random` y `Timer`.

## Nuevas características del IDE

La bandeja de componentes almacena los controles que no tienen una representación visual en los formularios.

## Resumen

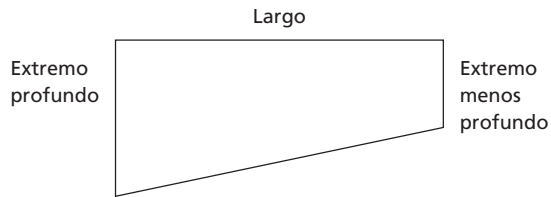
El sistema de C# tiene una gran variedad de clases que podemos (y deberíamos) utilizar. Además de las clases de controles que se encuentran en el cuadro de herramientas, existen otras que podemos incorporar a nuestros programas mediante la palabra clave `using` y el constructor apropiado.

## EJERCICIOS

- 6.1 Coloque una barra de seguimiento en un formulario, junto con dos cuadros de texto y un botón. Al hacer clic en el botón deberán establecerse las propiedades `Minimum` y `Maximum` de la barra de seguimiento, con base en los números introducidos en los cuadros de texto. Cuando el usuario desplace la barra de seguimiento deberán aparecer los valores de las propiedades `Minimum` y `Maximum` en cuadros de mensaje.
- 6.2 Escriba un programa que comience desplegando el número 1 en una etiqueta. Al hacer clic en un botón deberá incrementarse el valor. Utilice una variable privada inicializada en 1, y establezca el valor de la etiqueta en el constructor.
- 6.3 Escriba un programa que produzca un número aleatorio entre 200 y 400 cada vez que se haga clic en un botón. El programa deberá mostrar ese número junto con la suma y el promedio de todos los números recibidos hasta ese momento. A medida que usted oprima el botón repetidamente, el promedio deberá llegar a 300. Si no lo hace podemos sospechar que la responsabilidad es del generador de números aleatorios. Después de todo, ¡lo mismo ocurriría si lanzáramos una moneda al aire cien veces seguidas y siempre cayera cara!
- 6.4 (a) Escriba un programa que convierta grados Celsius (centígrados) en Fahrenheit. El usuario tendrá que introducir el valor Celsius en un cuadro de texto (utilice valores enteros). Al hacer clic en un botón deberá aparecer el valor Fahrenheit equivalente en una etiqueta. La fórmula de conversión es:
 
$$f = (c * 9) / 5 + 32;$$
 (b) Modifique el programa de manera que se introduzca el valor Celsius mediante una barra de seguimiento, cuyos valores mínimo y máximo serán 0 y 100, respectivamente.  
 (c) Represente ambas temperaturas mediante rectángulos largos y delgados en un cuadro de imagen.
- 6.5 Escriba un programa que calcule el volumen de una alberca y muestre el plano transversal de la misma en un cuadro de imagen. El ancho de la alberca está fijo en 5 metros, y su largo lo está en 20 metros. El programa debe tener dos barras de seguimiento: una para ajustar la profundidad del extremo más hondo, y otra para hacer lo propio respecto del extremo menos profundo. La profundidad mínima de cada extremo debe ser de 1 metro. Seleccione valores apropiados para las propiedades `Minimum` y `Maximum` de la barra de seguimiento en tiempo de diseño. La fórmula para determinar el volumen es:

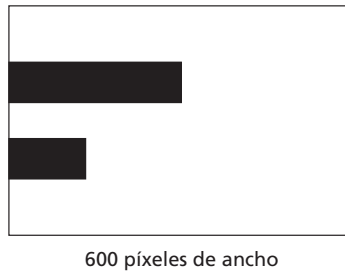
$$v = \text{profundidadPromedio} * \text{ancho} * \text{largo};$$

En la Figura 6.5 se muestra un plano transversal de la alberca.



**Figura 6.5** Plano transversal de una alberca.

- 6.6** Escriba un programa que despliegue un avance de minutos y segundos, representándolos mediante dos rectángulos largos: haga que el ancho máximo de los rectángulos sea de 600 píxeles para simplificar la aritmética (10 píxeles por cada minuto y cada segundo). Haga que los dos rectángulos se redibujen cada segundo. La Figura 6.6 muestra una representación de 30 minutos y 15 segundos.



**Figura 6.6** Visualización de tiempo: 30 minutos, 15 segundos.

El programa debe realizar el conteo en segundos mediante un temporizador, y desplegar el total además del tiempo transcurrido en minutos y segundos. Recuerde que, dado un número total de segundos, podemos utilizar el operador % para agruparlo en minutos enteros y segundos restantes. Con el propósito de agilizar la prueba del programa, reduzca el intervalo de tiempo de 1000 a 200 milisegundos, por ejemplo.

- 6.7** Este ejercicio le permitirá comprender cómo se escribe un juego de geometría:
- Escriba un programa con dos barras de seguimiento que controlen la posición horizontal y vertical de un círculo con 200 píxeles de diámetro.
  - Agregue una tercera barra de seguimiento para controlar el diámetro del círculo.
  - Lo que sigue es un juego basado en el hecho matemático de que se puede dibujar un círculo con base en tres puntos cualesquiera. El programa debe mostrar tres puntos (cada uno de ellos es un pequeño círculo relleno) al hacer clic en un botón llamado "Siguiente juego". Algunas posiciones iniciales convenientes son (100, 100), (200, 200) y (200, 100), pero puede agregar un pequeño número aleatorio a estas posiciones para obtener más variedad. El jugador debe manipular el círculo hasta que considere que éste pasa por cada uno de los puntos; después tendrá que hacer clic en un botón denominado "Listo".
  - Agregue un temporizador para mostrar cuánto tiempo le lleva al usuario realizar la tarea.

**SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**

- 6.1** El programa se compilará y ejecutará de todas formas, pero es muy probable que produzca resultados incorrectos. Ahora modifica el valor de una variable compartida entre los métodos; antes modificaba una variable local.
- 6.2** El programa trata de acceder a una etiqueta antes de que ésta haya sido creada (en `InitializeComponent`). Esto produce un error en tiempo de ejecución.
- 6.3** No hay consecuencias graves. Las barras de seguimiento alteran su valor máximo con base en el tamaño del cuadro de imagen a medida que se ejecuta el programa.
- 6.4** Los métodos típicos son: avanzar a la siguiente pista, detener, iniciar. Las propiedades no son tan universales, pero muchos reproductores muestran el número de la pista en ejecución. Tanto los métodos como las propiedades son miembros.
- 6.5** `a` se convierte en el área de la etiqueta, en píxeles.  
La altura de la etiqueta se iguala con su ancho; en otras palabras, la etiqueta se hace cuadrada.
- 6.6** En esta analogía hay un constructor, al cual le pasamos un color. La instancia es el automóvil real que usted conduce (la fotografía del catálogo en realidad sólo es documentación que le muestra la apariencia que tendrá su automóvil).
- 6.7** Introducimos una variable que podría llamarse `segundaCuenta`. Esta variable se incrementa en el método `Tick` del temporizador. No puede ser local, ya que perdería su valor cuando el método terminara su ejecución. En vez de ello debe declararse como una variable de instancia, en la parte superior del programa. Utilizamos la división entera entre 60 para calcular el número de minutos.

```
public partial class Form1 : ...
{
    private int segundaCuenta = 0;
    private void timer1_Tick(...)
    {
        segundaCuenta = segundaCuenta + 1;
        label1.Text = Convert.ToString(segundaCuenta / 60);
    }
}
```

# 7

## Selección

**En este capítulo conoceremos cómo:**

- utilizar las instrucciones `if` y `switch` para llevar a cabo evaluaciones;
- utilizar los operadores de comparación, como `>`;
- utilizar los operadores lógicos `&&`, `||` y `!`;
- declarar y utilizar datos booleanos.

### ● Introducción

---

Todos los seres humanos hacemos selecciones en la vida diaria. Usamos un abrigo si llueve; compramos un CD si tenemos suficiente dinero. Las selecciones también se utilizan mucho en los programas. La computadora evalúa un valor y, de acuerdo con el resultado, toma un curso de acción u otro. Cada vez que un programa hace una selección entre varias acciones y decide realizar una u otra se utiliza una instrucción `if` o `switch` para describir la situación.

Ya hemos visto que los programas de computadora son series de instrucciones que esta última debe seguir. La computadora obedece las instrucciones una tras otra, en secuencia. Sin embargo, algunas veces desearíamos que se evalúen ciertos datos y se elija una de varias acciones dependiendo del resultado de la evaluación. Por ejemplo, tal vez necesitemos que la computadora evalúe la edad de una persona y le diga si puede votar o es demasiado joven para hacerlo. A esto se le conoce como *selección*, y para llevarla a cabo se utiliza una instrucción conocida como `if`, el tema central de este capítulo.

Las instrucciones `if` son tan importantes que se utilizan en todos los lenguajes de programación que se han inventado.

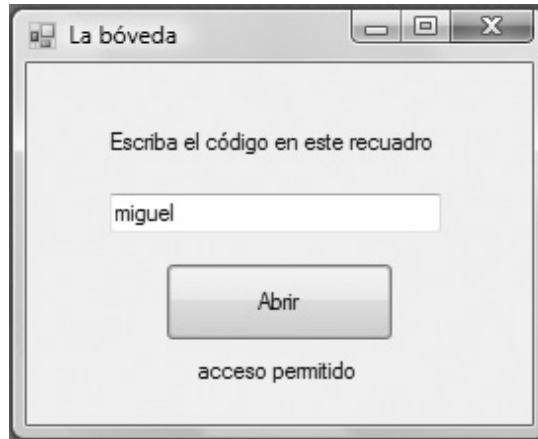


Figura 7.1 Interfaz del programa La bóveda.

## ● La instrucción `if`

Nuestro primer ejemplo es un programa que simula el candado digital de una bóveda de seguridad. En la Figura 7.1 se muestra la interfaz del mismo. La bóveda se mantiene cerrada hasta que el usuario introduce el código correcto en un cuadro de texto. Al principio este cuadro aparece vacío. El programa compara el texto introducido con el código correcto. Si son iguales se despliega un mensaje.

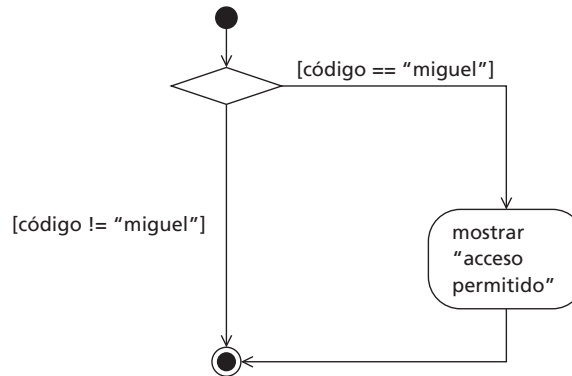
```
private void button1_Click(object sender, EventArgs e)
{
    string código;

    label2.Text = "";
    código = textBox1.Text;
    if (código == "miguel")
    {
        label2.Text = "acceso permitido";
    }
}
```

La instrucción `if` prueba el valor de la cadena de texto. Si ésta contiene el valor “miguel”, la instrucción que está entre los corchetes `{ y }` se realiza. Por otro lado, si la cadena es diferente de “miguel” la instrucción entre los corchetes es ignorada y se ejecuta cualquier otra instrucción que esté después.

Tenga en cuenta que la condición a evaluar se encierra entre paréntesis; ésta es una regla gramatical de C#. Observe también que la prueba de igualdad utiliza el operador `==` (no el operador `=`).





**Figura 7.2** Diagrama de acción de una instrucción `if`.

Una forma de comprender la función de las instrucciones `if` es mediante un diagrama de acción (Figura 7.2). Este diagrama muestra de manera gráfica la instrucción `if` del programa La bóveda. Para interpretarlo empiece en el círculo relleno de la parte superior izquierda y siga las flechas. La decisión está representada por el diamante, y las dos posibles condiciones se ilustran entre corchetes. La acción se muestra en el cuadro de esquinas redondeadas, y el fin de la secuencia está señalado por el círculo que se halla en la parte inferior izquierda del diagrama.

La instrucción `if` consta de dos partes:

- la condición a evaluar;
- la instrucción o secuencia de instrucciones a ejecutar si la condición es verdadera.

Todos los programas consisten en una secuencia de acciones; en el caso del ejemplo anterior dicha secuencia es:

1. Se recibe una pieza de texto del cuadro de texto.
2. Luego se realiza una evaluación.
3. Si la evaluación resulta positiva, se despliega un mensaje avisando que la bóveda está abierta.

Es muy frecuente que se lleve a cabo no sólo una acción, sino secuencias completas de acciones si el resultado de la evaluación es verdadero. En este caso, las acciones a realizar deben ir encerradas entre los corchetes.

## Organización del código

Observe que se aplica sangría a las líneas de código (esto significa que se utilizan espacios para desplazar el texto hacia la derecha) para reflejar la estructura de esta pieza del programa. El sistema de desarrollo de Microsoft realiza esto de manera automática cuando usted escribe una instrucción `if`. Pero si su programa se vuelve un enredo, como sucede a menudo al momento de codificar, puede hacer que el IDE aplique el formato apropiado. Para ello abra el menú **Edición** y seleccione la opción **Seleccionar todo**. Después elija **Edición, Avanzado, Dar formato a la selección**.

Aunque el uso de sangrías no es esencial, es muy conveniente para que quienes lean el código puedan entenderlo con facilidad. Todos los buenos programas (cualquiera que sea el lenguaje que se haya utilizado para crearlos) cuentan con esta característica de sangrado, y todos los buenos programadores la utilizan.

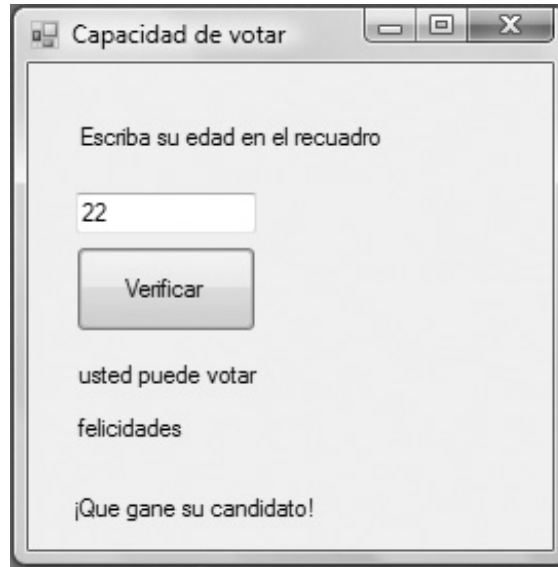


Figura 7.3 Interfaz del programa para revisar la capacidad de votar.

### ● if ... else

Algunas veces es necesario especificar *dos* secuencias de acciones: las que se llevarán a cabo si la condición es verdadera, y las que serán realizadas si es falsa.

El usuario del programa para verificar la capacidad de votar escribe su edad en un cuadro de texto y hace clic en un botón; el programa decide entonces si puede votar o no. La interfaz de este programa se muestra en la Figura 7.3. Cuando el usuario hace clic en el botón el programa extrae la información introducida en el cuadro de texto, convierte la cadena en un entero y coloca el número en la variable llamada `edad`. A continuación se necesita que el programa realice diferentes acciones, dependiendo de si el valor es:

- mayor que 17, o
- igual o menor que 17.

Luego se muestran los resultados de la evaluación en varias etiquetas.

```
private void button1_Click(object sender, EventArgs e)
{
    int edad;

    edad = Convert.ToInt32(textBox1.Text);
    if (edad > 17)
    {
        etiquetaDecisión.Text = "usted puede votar";
        etiquetaComentario.Text = "felicidades";
    }
}
```

```

else
{
    etiquetaDecisión.Text = "usted no puede votar";
    etiquetaComentario.Text = "lo siento";
}
etiquetaDespedida.Text = "¡Que gane su candidato!";
}

```

Esta instrucción `if` consta de tres partes:

- la condición a evaluar, en este caso, si la edad es superior a 17 años;
- la instrucción o secuencia de instrucciones que se ejecutarán si la condición es verdadera; esta parte debe ir encerrada entre corchetes;
- la instrucción o instrucciones a ejecutar si la condición es falsa; esta parte debe ir encerrada entre corchetes.

El nuevo elemento en este código es la palabra clave `else`, que introduce la segunda parte de la instrucción `if`. Observe de nuevo cómo la sangría ayuda a enfatizar la intención del programa.

Para comprender las instrucciones `if...else` resulta útil usar un diagrama de acción como el que se muestra en la Figura 7.4. Este diagrama ilustra la condición a evaluar y las dos acciones separadas.

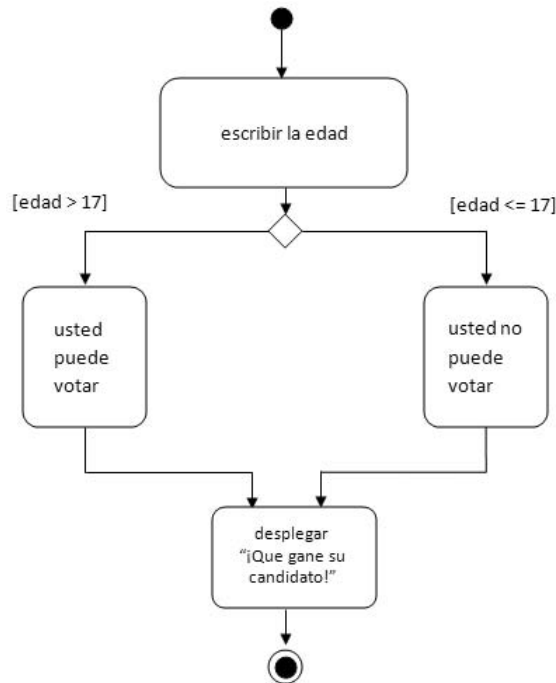


Figura 7.4 Diagrama de acción de una instrucción `if...else`.

## ● Operadores de comparación

Los programas que hemos venido utilizando como ejemplo emplean algunos operadores de comparación. A continuación le presentamos una lista completa de dichos operadores:

Símbolo	Significado
>	mayor que
<	menor que
==	igual que
!=	no es igual que
<=	menor o igual que
>=	mayor o igual que

Aquí podemos ver de nuevo que C# utiliza el signo “igual que” (==) para evaluar si dos elementos son iguales.

La elección del operador apropiado suele ser una tarea que debe realizarse con mucho cuidado. En el programa para evaluar la capacidad de votar probablemente la evaluación apropiada sería:

```
if (edad >= 18)
{
    etiquetaDecisión.Text = "usted puede votar";
}
```

Tenga en cuenta que por lo general es posible escribir condiciones en una de dos formas. Los siguientes dos fragmentos de código obtienen exactamente el mismo resultado, pero utilizan distintas condiciones. El código:

```
if (edad >= 18)
{
    etiquetaDecisión.Text = "usted puede votar";
}
else
{
    etiquetaDecisión.Text = "lo siento";
}
```

obtiene el mismo resultado que:

```
if (edad < 18)
{
    etiquetaDecisión.Text = "lo siento";
}
else
{
    etiquetaDecisión.Text = "usted puede votar";
}
```

Aunque estos dos fragmentos logren el mismo resultado final probablemente el primero sea mejor, debido a que establece con más claridad la condición para poder votar.

**PRÁCTICA DE AUTOEVALUACIÓN**

**7.1** ¿Las siguientes dos piezas de código de C# obtienen el mismo resultado o no?

```
if (edad > 18)
{
    etiquetaDecisión.Text = "usted puede votar";
}

if (edad < 18)
{
    etiquetaDecisión.Text = "usted no puede votar";
}
```

En el siguiente programa crearemos dos barras de seguimiento usando el cuadro de herramientas, y desplegaremos círculos de tamaño equivalente (Figura 7.5). El programa comparará los valores e informará cuál de ellos tiene un valor superior. Para ello se utiliza el método de biblioteca `FillEllipse`, mediante el cual se dibujará un círculo sólido cuyo diámetro sea igual al valor que se obtiene de la barra de seguimiento correspondiente.

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    CompararValores();
}

private void trackBar2_Scroll(object sender, EventArgs e)
{
    CompararValores();
}
```



Figura 7.5 El programa ¿Cuál es mayor?

```
private void CompararValores()
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    SolidBrush miPincelRojo = new SolidBrush(Color.Red);
    SolidBrush miPincelAzul = new SolidBrush(Color.Blue);

    int valorRojo, valorAzul;
    valorRojo = trackBar1.Value;
    valorAzul = trackBar2.Value;

    papel.Clear(Color.White);
    papel.FillEllipse(miPincelRojo, 10, 10, valorRojo, valorRojo);
    papel.FillEllipse(miPincelAzul, 80, 10, valorAzul, valorAzul);

    if (valorRojo > valorAzul)
    {
        label1.Text = "el círculo rojo es mayor";
    }
    else
    {
        label1.Text = "el círculo azul es mayor";
    }
}
```

Este programa funciona bien, pero ilustra nuevamente la importancia de tener cuidado al usar instrucciones `if`. ¿Qué ocurre cuando los valores de ambos círculos son iguales? El programa determina que el azul es mayor, aunque esto en realidad no es así. Podríamos mejorar el programa para establecer las condiciones con más claridad, cambiando la instrucción `if` por el siguiente código:

```
if (valorRojo > valorAzul)
{
    label1.Text = "el círculo rojo es mayor";
}
if (valorAzul > valorRojo)
{
    label1.Text = "el círculo azul es mayor";
}
if (valorRojo == valorAzul)
{
    label1.Text = "Los círculos son iguales";
}
```

El siguiente ejemplo es un programa que lleva el registro del valor más grande de una cifra a medida que ésta va cambiando. Algunos amplificadores estereofónicos tienen una pantalla con gráficos de barras en donde se muestra el volumen de salida. El nivel de la pantalla aumenta y disminuye de acuerdo con el volumen en un momento dado. De igual manera, la pantalla de algunos de esos

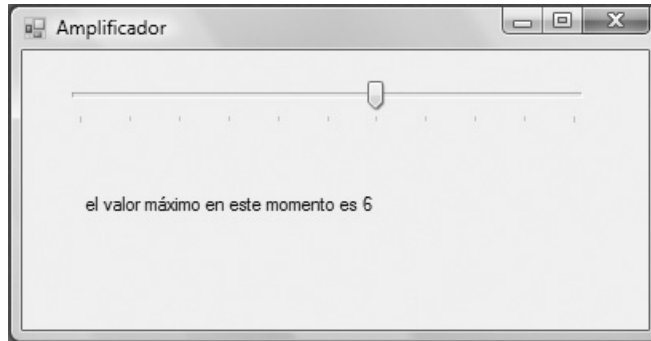


Figura 7.6 El programa Amplificador.

aparatos tiene un indicador que muestra el valor máximo de salida en un momento dado. Por su parte, nuestro programa despliega el valor numérico máximo en el que se encuentra la barra de seguimiento (vea la Figura 7.6), para lo cual utiliza una sola instrucción `if` que compara el valor actual de la barra con el de `max`, una variable a nivel de clase que contiene el mayor valor obtenido en un momento dado. La variable `max` se declara así:

```
private int max = 0;
```

y el método para manejar los eventos de la barra de seguimiento es:

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    int volumen;

    volumen = trackBar1.Value;
    if (volumen > max)
    {
        max = volumen;
    }
    label1.Text = "el valor máximo es " + Convert.ToString(max);
}
```

## PRÁCTICA DE AUTOEVALUACIÓN

**7.2** Escriba un programa que muestre el valor numérico mínimo en el que se encuentre la barra de seguimiento.

A continuación revisaremos un programa que simula la acción de tirar dos dados. La computadora decide al azar el valor que mostrarán los dados. Debemos crear un botón llamado “Tirar”. Al hacer clic en él, el programa obtendrá dos números al azar y los utilizará como los valores de los dados (Figura 7.7).



Figura 7.7 Juego de azar.

Para obtener un número aleatorio creamos un objeto de la clase de biblioteca **Random**, y después utilizamos su método **Next**. Este método devuelve un número aleatorio, un valor **int** en cualquier rango que seleccionemos y hayamos especificado mediante los parámetros. En el capítulo 6 vimos una introducción a esta clase.

El código del programa Juego de azar se muestra a continuación. En él es necesario declarar la siguiente variable a nivel de clase:

```
private Random númeroAleatorio = new Random();
```

y el método para el manejo de eventos es:

```
private void button1_Click(object sender, EventArgs e)
{
    int dado1, dado2;

    dado1 = númeroAleatorio.Next(1, 6);
    dado2 = númeroAleatorio.Next(1, 6);

    label1.Text = "los valores de los dados son "
        + Convert.ToString(dado1) + " y " + Convert.ToString(dado2);
    if (dado1 == dado2)
    {
        label2.Text = "los dados son iguales - usted gana";
    }
    else
    {
        label2.Text = "los dados no son iguales - usted pierde";
    }
}
```



## ● And, or, not

En programación es muy frecuente que necesitemos evaluar dos cosas a la vez. Suponga, por ejemplo, que tenemos que evaluar si alguien debe pagar una tarifa reducida por un boleto:

```
if (edad > 6 && edad < 16)
{
    label1.Text = "tarifa infantil";
}
```

Los caracteres `&&` representan uno de los operadores lógicos de C#; equivale a la conjunción “y”.

Es posible utilizar paréntesis adicionales para mejorar la legibilidad de estas condiciones más complejas. Por ejemplo, podemos replantear la instrucción anterior de la siguiente manera:

```
if ((edad > 6) && (edad < 16))
{
    label1.Text = "tarifa infantil";
}
```

Aunque los paréntesis internos no son esenciales, sirven para diferenciar las dos condiciones que se están evaluando.

Podría verse tentado a escribir:

```
if (edad > 6 && < 16) // ¡error!
```

pero eso sería incorrecto, ya que las condiciones se tienen que establecer por completo, como se muestra a continuación:

```
if (edad > 6 && edad < 16) // esta instrucción es correcta
```

Por otro lado, podríamos utilizar el operador `||` (equivalente a la disyunción “o”) en una instrucción `if` de la siguiente forma:

```
if (edad < 6 || edad > 60)
{
    label1.Text = "tarifa reducida";
}
```

En este caso la tarifa reducida se aplica cuando las personas son menores de seis años o mayores de sesenta.

El operador `!` equivale a la palabra “no”, y se utiliza mucho en programación, aun cuando el uso de negativos no siempre es todo lo claro que pudiera desearse. He aquí un ejemplo de su uso:

```
if (! (edad > 16))
{
    label1.Text = "demasiado joven";
}
```

Esto significa que la evaluación busca establecer si una persona en particular es mayor de 16 años. Si el resultado es verdadero, el operador `!` lo convertirá en falso, y si es falso lo hará verdadero. Cuando el resultado se valide como verdadero, se desplegará el mensaje especificado. Esto, desde luego, puede escribirse de manera más simple sin el operador `!`.

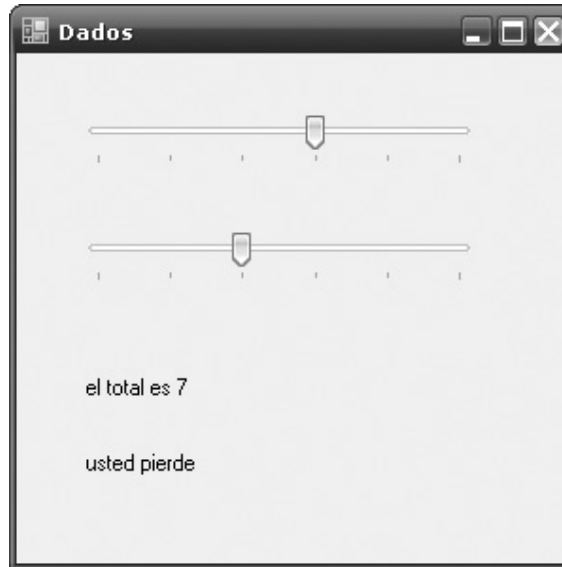


Figura 7.8 El programa de los dados.

### PRÁCTICA DE AUTOEVALUACIÓN

#### 7.3 Rediseñe la instrucción `if` anterior sin utilizar el operador `!`.

El siguiente programa ilustra una serie de evaluaciones más compleja. Se lanzan dos dados en un juego de azar, y el programa tiene que decidir cuál es el resultado. Crearemos dos barras de seguimiento, cada una con un rango de 1 a 6, para especificar los valores de los dos dados correspondientes (Figura 7.8). Para empezar usaremos la regla según la cual sólo una puntuación total de 6 gana.

El código del programa se muestra a continuación. Cada vez que se mueve una de las dos barras de seguimiento se invoca el método para mostrar el valor total y decidir si el jugador ganó.

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    VerificarValores();
}

private void trackBar2_Scroll(object sender, EventArgs e)
{
    VerificarValores();
}

private void VerificarValores()
{
    int dado1, dado2, total;
```

```

        dado1 = trackBar1.Value;
        dado2 = trackBar2.Value;
        total = dado1 + dado2;
        label1.Text = "el total es " + Convert.ToString(total);
        if (total == 6)
        {
            label2.Text = "usted gana";
        }
        else
        {
            label2.Text = "usted pierde";
        }
    }
}

```

A continuación alteraremos las reglas para ver cómo rediseñar el programa. Suponga que cualquier par de valores idénticos gana; por ejemplo, los dos dados con un punto, con dos puntos, etc. En ese caso la instrucción `if` sería:

```

if (dado1 == dado2)
{
    label2.Text = "usted gana";
}

```

Pero también podríamos decidir que el jugador sólo gana si obtiene un total de 2 o un total de 7:

```

if ((total == 2) || (total == 7))
{
    label2.Text = "usted gana";
}

```

Observe de nuevo que hemos encerrado cada una de las condiciones entre paréntesis. Esta práctica no es estrictamente necesaria en C#, pero ayuda mucho a aclarar el significado de la condición a evaluar.

En la tabla siguiente se presentan todos los operadores que hemos comentado en la sección:

Símbolo	Significado
&&	and; equivale a la conjunción "y"
	or; equivale a la disyunción "o"
!	not; equivale a la negación "no"

#### PRÁCTICAS DE AUTOEVALUACIÓN

- 7.4** Modifique el programa de los dados para que el jugador gane cuando obtenga un valor total de 2, de 5 o de 7.
- 7.5** Escriba instrucciones `if` para evaluar si alguien puede obtener un empleo de tiempo completo. La regla es que debe tener 16 años o más, y ser menor de sesenta y cinco.

## ● Instrucciones `if` anidadas

---

Analice el siguiente fragmento de código de un programa:

```
if (edad > 6)
{
    if (edad < 16)
    {
        label1.Text = "tarifa junior";
    }
    else
    {
        label1.Text = "tarifa de adulto";
    }
}
else
{
    label1.Text = "tarifa infantil";
}
```

Aquí podemos ver que la segunda instrucción `if` está completamente dentro de la primera (la sangría contribuye a mejorar la legibilidad). A esto se le conoce como *anidamiento*. Anidar no es lo mismo que sangrar el código, pero el sangrado hace el anidamiento muy evidente.

El efecto general de esta pieza de código es:

- Si la persona es mayor de 6 años y menor de 16, pagará la tarifa junior.
- Si la persona es mayor de 6 años pero no menor de 16, pagará la tarifa de adulto.
- Si la persona no es mayor de 6 años, pagará la tarifa infantil.

Es común ver el anidamiento en los programas, pero cuando esto ocurre, el código tiene una complejidad que dificulta un poco su comprensión. Muchas veces es posible escribir un programa de manera más simple, utilizando los operadores lógicos. Por ejemplo, en el código siguiente se obtiene el mismo resultado que en el caso anterior sin utilizar el anidamiento:

```
if (edad >= 16)
{
    label1.Text = "tarifa de adulto";
}
if (edad <= 6)
{
    label1.Text = "tarifa infantil";
}
if ((edad > 6) && (edad < 16))
{
    label1.Text = "tarifa junior";
}
```

Aquí tenemos dos piezas de código que obtienen el mismo resultado, uno con anidamiento y el otro aprovechando los operadores lógicos. Algunas personas argumentan que es difícil entender el anida-

miento, lo cual provoca que el programa sea propenso a errores; en consecuencia, sería mejor evitarlo y beneficiarnos directamente de los operadores lógicos.

#### PRÁCTICAS DE AUTOEVALUACIÓN

**7.6** Escriba un programa que reciba como información de entrada una cantidad que represente un rango de salario; utilice para ello una barra de seguimiento y determine qué monto de impuestos deben pagar los usuarios de acuerdo con las siguientes reglas:

El usuario no tiene que pagar impuestos si gana hasta \$10,000; debe pagar 20% de impuestos si gana más de \$10,000 y hasta \$50,000; Por último, debe pagar 90% de impuestos si gana más de \$50,000. La barra de seguimiento debe tener un rango de 0 a 100,000.

**7.7** Escriba un programa que conste de tres barras de seguimiento y muestre el mayor de los tres valores representados en ellas al hacer clic en un botón.

**7.8** La agencia de viajes "Joven y bella" sólo acepta clientes entre los 18 y los 30 años (el criterio se basa en que si el cliente es menor de 18 años no tiene dinero, y si es mayor de 30 tiene demasiadas arrugas). Escriba un programa para evaluar si usted puede utilizar los servicios de esta empresa para planear sus vacaciones.

### ● La instrucción `switch`

Esta instrucción constituye otra forma de usar muchas instrucciones `if`. Usted podrá realizar todo lo que necesite con la ayuda de las instrucciones `if`, pero `switch` puede ser una alternativa más eficiente en circunstancias apropiadas. Por ejemplo, suponga que necesitamos una pieza de código para mostrar el día de la semana como una cadena de texto. Imagine que el programa representa el día de la semana como una variable `int` llamada `númeroDía` con uno de los valores del 1 al 7 para representar los días de lunes a domingo. Queremos convertir la versión numérica del día en una versión de cadena de texto llamada `nombreDía`. Para ello podríamos escribir la siguiente serie de instrucciones `if`:

```
if (númeroDía == 1)
{
    nombreDía = "Lunes";
}
if (númeroDía == 2)
{
    nombreDía = "Martes";
}
if (númeroDía == 3)
{
    nombreDía = "Miércoles";
}
if (númeroDía == 4)
{
    nombreDía = "Jueves";
}
```

```
if (númeroDía == 5)
{
    nombreDía = "Viernes";
}
if (númeroDía == 6)
{
    nombreDía = "Sábado";
}
if (númeroDía == 7)
{
    nombreDía = "Domingo";
}
```

Aunque la pieza de código que acabamos de proponer es clara y está bien estructurada, hay una alternativa que cumple la misma función utilizando la instrucción `switch`:

```
switch (númeroDía)
{
    case 1:
        nombreDía = "Lunes";
        break;

    case 2:
        nombreDía = "Martes";
        break;

    case 3:
        nombreDía = "Miércoles";
        break;

    case 4:
        nombreDía = "Jueves";
        break;

    case 5:
        nombreDía = "Viernes";
        break;

    case 6:
        nombreDía = "Sábado";
        break;

    case 7:
        nombreDía = "Domingo";
        break;
}
```

La instrucción `break` transfiere el control al final de la instrucción `switch`, el cual está marcado con una llave de cierre. Esto nos permite ver con más claridad lo que se va a hacer, en contraste con la serie de instrucciones `if` equivalente.

Las instrucciones `switch` como la anterior resultan más comprensibles mediante un diagrama de acción, como el que se muestra en la Figura 7.9.

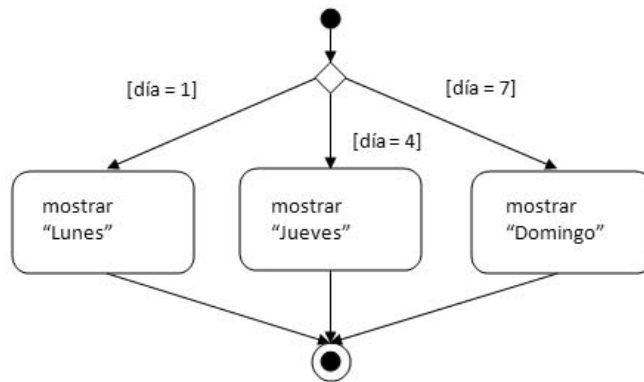


Figura 7.9 Diagrama de acción que ilustra parte de una instrucción `switch`.

### PRÁCTICA DE AUTOEVALUACIÓN

**7.9** Escriba un método que convierta los enteros 1, 2, 3 y 4 en las palabras diamantes, corazones, tréboles y picas, respectivamente.

Puede haber varias instrucciones en cada una de las opciones de una instrucción `switch`. Por ejemplo, una de las opciones podría ser:

```

case 6:
    MessageBox.Show("hurra");
    nombreDía = "Sábado";
    break;
  
```

Otra característica de la instrucción `switch` consiste en agrupar varias opciones, como en el siguiente ejemplo:

```

switch (númeroDía)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        nombreDía = "día laboral";
        break;

    case 6:
    case 7:
        nombreDía = "fin de semana";
        break;
}
  
```

La opción `default` (predeterminada) es otra parte de la instrucción `switch` que sirve para ciertos casos. Siguiendo con el ejemplo anterior, suponga que el valor del entero que representa el día de la semana se introduce mediante un cuadro de texto. En este caso existe claramente la posibilidad de que el usuario introduzca por error un número que no se encuentre en el rango de 1 a 7. Cualquier código decente necesita tener esto en cuenta para poder evitar que ocurra algo extraño, o que el programa falle. La instrucción `switch` es capaz de lidiar con esta situación, ya que podemos proveer una opción “atrapa todo” (predeterminada) para que entre en acción si ninguna de las demás alternativas es válida:

```
switch (númeroDía)
{
    case 1:
        nombreDía = "Lunes";
        break;

    case 2:
        nombreDía = "Martes";
        break;

    case 3:
        nombreDía = "Miércoles";
        break;

    case 4:
        nombreDía = "Jueves";
        break;

    case 5:
        nombreDía = "Viernes";
        break;

    case 6:
        nombreDía = "Sábado";
        break;

    case 7:
        nombreDía = "Domingo";
        break;

    default:
        nombreDía = "día ilegal";
        break;
}
```

Si no se escribe una opción `default` como parte de una instrucción `switch` y ninguno de los casos provistos corresponde al valor actual de la variable, el resultado es que se ignorarán todas las opciones.

La instrucción `switch` es muy útil, pero por desgracia no es tan flexible como podría. Suponga, por ejemplo, que deseamos escribir un programa para mostrar dos números: primero el mayor y después el menor. Si empleáramos instrucciones `if` el código quedaría así:



```

if (a > b)
{
    label1.Text = Convert.ToString(a) + " es mayor que "
                + Convert.ToString(b);
}
if (b > a)
{
    label1.Text = Convert.ToString(b) + " es mayor que "
                + Convert.ToString(a);
}
if (a == b)
{
    label1.Text = "son iguales";
}

```

Podríamos vernos tentados a replantear este programa utilizando una instrucción `switch`, como se muestra a continuación:

```

switch (?) // ¡cuidado! código inválido en C#
{
    case a > b:
        label1.Text = Convert.ToString(a) + " es mayor que"
                    + Convert.ToString(b);

        break;

    case b > a:
        label1.Text = Convert.ToString(b) + " es mayor que"
                    + Convert.ToString(a);

        break;

    case a == b:
        label1.Text = "son iguales";

        break;
}

```

El problema es que esto no se permite, ya que, según lo indicado por el signo de interrogación, `switch` sólo trabaja con variables enteras o de cadena, y `case` no puede utilizar los operadores `>`, `==`, `<`, etc.

## ● Variables booleanas

Todos los tipos de variables que hemos visto hasta ahora están diseñados para contener números o cadenas de texto. En esta sección conoceremos un nuevo tipo de variable, llamado `bool`, que sólo puede contener los valores `true` (verdadero) o `false` (falso). Los términos `bool`, `true` y `false` son palabras reservadas en C#, por lo cual no pueden ser utilizadas para ningún otro fin. Este tipo de variable recibió su nombre en honor al matemático inglés del siglo XIX George Boole, quien hizo una gran contribución al desarrollo de la lógica matemática, en donde las ideas de verdadero y falso desempeñan un papel central.

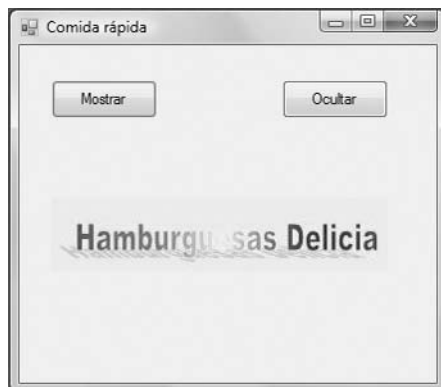


Figura 7.10 El anuncio de comida rápida.

Analicemos las propiedades de las etiquetas disponibles en el cuadro de herramientas para comprender cómo funcionan las variables booleanas. La Figura 7.10 muestra la interfaz de un programa que despliega u oculta el nombre de un restaurante de comida rápida mediante el uso de botones. Cuando el usuario hace clic en los botones se modifica la propiedad `visible` de la etiqueta:

```
private void botónMostrar_Click(object sender, EventArgs e)
{
    label1.Visible = true;
}

private void botónOcultar_Click(object sender, EventArgs e)
{
    label1.Visible = false;
}
```

La propiedad `visible` es de tipo `bool` y nos permite cambiar su valor, tal como se ilustra en el código del ejemplo anterior. También podemos evaluar su valor mediante el uso de instrucciones `if`. Para demostrarlo podríamos rediseñar el programa de manera que tenga un solo botón que despliegue u oculte el anuncio a voluntad; esto se logra mediante el uso de las instrucciones siguientes:

```
private void button1_Click(object sender, EventArgs e)
{
    if (label1.Visible == true)
    {
        label1.Visible = false;
    }
    else
    {
        label1.Visible = true;
    }
}
```

Ya hemos visto cómo utilizar una propiedad `bool`; veamos ahora cómo declarar nuestras propias variables booleanas. Es posible declarar una variable de tipo `bool` de la siguiente forma:

```
bool finalizado;
```

Además, podemos asignarle cualquiera de los valores `true` y `false`, como en el siguiente ejemplo:

```
finalizado = true;
```

También podemos evaluar el valor de una variable `bool` en una instrucción `if`. Por ejemplo:

```
if (finalizado)
{
    MessageBox.Show("Adiós");
}
```

En este caso se evalúa el valor de `finalizado` y, si es `true`, se ejecuta la instrucción que está dentro de las llaves. Una manera equivalente pero un poco menos eficiente de escribir la misma evaluación sería:

```
if (finalizado == true)
{
    MessageBox.Show("Adiós");
}
```

En programación, las variables booleanas se utilizan para recordar algo, tal vez durante un periodo corto de tiempo, o quizá durante todo el tiempo que el programa esté en ejecución. Como ejemplo veamos un programa que dibuja un rectángulo en un cuadro de imagen (Figura 7.11) al hacer

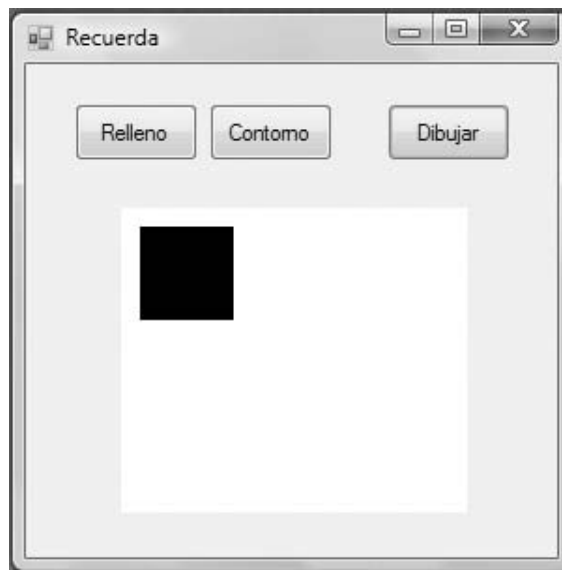


Figura 7.11 Interfaz del programa Recuerdo.

clic en un botón. Algunas veces queremos que sólo se trace el contorno del rectángulo, y otras que aparezca relleno. Hay que proveer dos botones para permitir que el usuario especifique su opción. Una vez que haga clic en uno de los botones, el programa recordará la opción hasta que el usuario haga clic de nuevo en cualquiera de los botones.

La declaración a nivel de clase de la variable `bool` es:

```
private bool relleno = true;
```

Ésta es la variable que “recuerda” lo que el usuario ha especificado. Puede tener el valor `true` (en cuyo caso la instrucción es que el rectángulo debe aparecer relleno) o el valor `false`.

Los métodos para manejar los clics de botón simplemente hacen que la variable `bool` sea `true` o `false`, según sea apropiado:

```
private void botónRelleno_Click(object sender, EventArgs e)
{
    relleno = true;
}

private void botónContorno_Click(object sender, EventArgs e)
{
    relleno = false;
}
```

Cuando el usuario hace clic en el botón “Dibujar”, lo primero que hace el programa es evaluar el valor de la variable `bool` llamada `relleno`, para lo cual utiliza una instrucción `if`. Después, con base en el valor de esa variable dibuja un rectángulo relleno o sólo su contorno.

```
private void botónDibujar_Click(object sender, EventArgs e)
{
    Graphics papel = pictureBox1.CreateGraphics();
    Pen miLápiz = new Pen(Color.Black);
    SolidBrush miPincel = new SolidBrush(Color.Black);
    papel.Clear(Color.White);
    if (relleno == true)
    {
        papel.FillRectangle(miPincel, 10, 10, 50, 50);
    }
    else
    {
        papel.DrawRectangle(miLápiz, 10, 10, 50, 50);
    }
}
```

Los métodos pueden usar valores booleanos como parámetros y como valores de retorno. Por ejemplo, este método verifica si tres números están en orden:

```
private bool EnOrden(int a, int b, int c)
{
    if ((a <= b) && (b <= c))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

## Fundamentos de programación

Por lo general la computadora lleva a cabo las instrucciones en secuencia, una por una. Las instrucciones `if` instruyen a la computadora para que evalúe el valor de ciertos datos y después elija una de varias acciones dependiendo del resultado de la evaluación. A este proceso de elección se le conoce como *selección*. A la evaluación de los datos se le denomina *condición*. Una vez que se completa una instrucción `if`, la computadora sigue llevando a cabo las instrucciones en secuencia.

## Errores comunes de programación

La condición dentro de una instrucción `if` debe ir encerrada entre paréntesis. Por ejemplo:

```
if (a > b)
```

Si desea evaluar una condición de igualdad debe utilizar el operador `==`, como en el siguiente ejemplo:

```
if (a == b)
```

Esta instrucción es completamente correcta:

```
if (código == "juan")
    label12.Text = "acceso autorizado";
```

aun cuando faltan las llaves que debieran rodear la instrucción. La regla de C# establece que si se debe ejecutar *una sola* instrucción, no son necesarias las llaves. Sin embargo, esto puede provocar molestos problemas de programación, por lo que le recomendamos insertar las llaves en todo momento.

Podría darse el caso de que escribiera una instrucción `if` de la siguiente forma:

```
if (a > 18 && < 25)
```

lo cual es incorrecto, ya que el operador `&&` debe enlazar dos condiciones completas, preferentemente encerradas entre paréntesis, como se muestra a continuación:

```
if ((a > 18) && (a < 25))
```

## Secretos de codificación

El primer tipo de instrucción `if` tiene la estructura:

```
if (condición)
{
    Instrucciones
}
```

El segundo tipo de instrucción `if` sigue esta estructura:

```
if (condición)
{
    Instrucciones
}
else
{
    Instrucciones
}
```

La instrucción `switch` tiene la siguiente estructura:

```
switch (variable)
{
    case valor1:
        instrucciones
        break;
    case valor2:
        instrucciones
        break;
    default:
        instrucciones
        break;
}
```

La acción predeterminada (`default`) es opcional.

## Nuevos elementos del lenguaje

- Estructuras de control para selección o toma de decisiones:

```
if, else
switch, case, break, default
```

- Los operadores de comparación `>`, `<`, `==`, `!=`, `<=` y `>=`.
- Los operadores lógicos `&&`, `||` y `!`.
- Las variables declaradas como `bool`, que pueden contener los valores `true` o `false`.

## Resumen

- Las instrucciones `if` permiten que el programador controle la secuencia de acciones al hacer que el programa lleve a cabo una evaluación. Después de realizar la evaluación, la computadora realiza una de varias acciones alternativas.
- Las instrucciones `if` presentan dos variedades:

```
if
if...else
```

- La instrucción `switch` proporciona una conveniente manera de llevar a cabo varias evaluaciones. Sin embargo, está restringida a realizar evaluaciones con números enteros o cadenas de texto.
- Las variables `bool` pueden contener el valor `true` o el valor `false`. Además, pueden evaluarse mediante instrucciones `if`.

## EJERCICIOS

- 7.1 Repartiendo cartas** Escriba un programa con un solo botón de manera que, cuando se haga clic en él, seleccione al azar una carta. Primero utilice el generador de números aleatorios de la biblioteca para crear un número en el rango de 1 a 4. Después convierta el número en un palo (corazón, diamante, trébol y pica). Luego utilice de nuevo el generador de números aleatorios para crear un número aleatorio en el rango de 1 a 13. Convierta el número en un as, un 2, 3, 5, etc., y finalmente muestre el valor de la carta elegida (sugerencia: utilice `switch` según sea apropiado).
- 7.2 Ordenamiento** Escriba un programa que reciba números de tres barras de seguimiento o tres cuadros de texto y los muestre en orden, de menor a mayor.
- 7.3 Entradas al cine** Escriba un programa para averiguar cuánto paga una persona por asistir a una función de cine. El programa debe recibir una edad a partir de una barra de seguimiento o cuadro de texto, y después tomar una decisión con base en estas condiciones:
- si el espectador es menor de 5 años, la función es gratis;
  - si tiene de 5 a 12 años, paga la mitad de la tarifa;
  - si tiene de 13 a 54 años, paga la tarifa completa;
  - si es mayor de 55, la función es gratis.
- 7.4 Apuestas** Un grupo de personas desean apostar sobre el resultado de tres lanzamientos de dados. Cada una de ellas debe apostar \$1 para tratar de adivinar el resultado de los tres lanzamientos. Escriba un programa que utilice el método de números aleatorios para simular tres lanzamientos de un dado, y que muestre las ganancias de acuerdo con las siguientes reglas:
- si los tres dados caen en seis: la persona gana \$20;
  - si los tres dados caen en el mismo número (pero no en seis): gana \$10;
  - si dos de tres dados cayeron en el mismo número: gana \$5.

- 7.5 Bóveda de combinación digital** Escriba un programa que actúe como un candado de combinación digital para una bóveda. Cree tres botones que representen los números 1, 2 y 3. El usuario tiene que hacer clic en los botones para tratar de adivinar los números correctos (por ejemplo, 331121). El programa debe permanecer inactivo hasta que se opriman los botones correctos. Cuando esto suceda mostrará un mensaje de felicitación al usuario. Debe haber un botón para reiniciar.

Mejore el programa de manera que tenga otro botón para que el usuario pueda modificar la combinación de la bóveda.

- 7.6 Juego de piedra, papel o tijera** En su forma original, dos jugadores eligen al mismo tiempo piedra, papel o tijera. La piedra gana a la tijera, el papel gana a la piedra, y la tijera gana al papel. Si ambos jugadores eligen la misma opción, es un empate. Escriba un programa para practicar este juego. El jugador debe seleccionar uno de tres botones marcados como piedra, papel o tijera. La computadora tendrá que realizar su elección al azar mediante el uso del generador de números aleatorios; también debe decidir y mostrar quién gana.

- 7.7 La calculadora** Escriba un programa que simule una calculadora de escritorio sencilla (Figura 7.12) y opere con números enteros. Debe tener un botón para cada uno de los 10 dígitos, del 0 al 9. También tendrá que contar con un botón para sumar y otro para restar, uno más para borrar el contenido de la “pantalla”, y un botón de igual (=) para obtener las respuestas.

Al oprimir el botón para borrar el contenido de la pantalla, ésta debe quedar en cero y el total (oculto) debe quedar también en cero.

Al hacer clic en el botón de cualquier dígito, éste deberá agregarse a la derecha de los dígitos que ya se encuentren en pantalla (en caso de haber alguno).

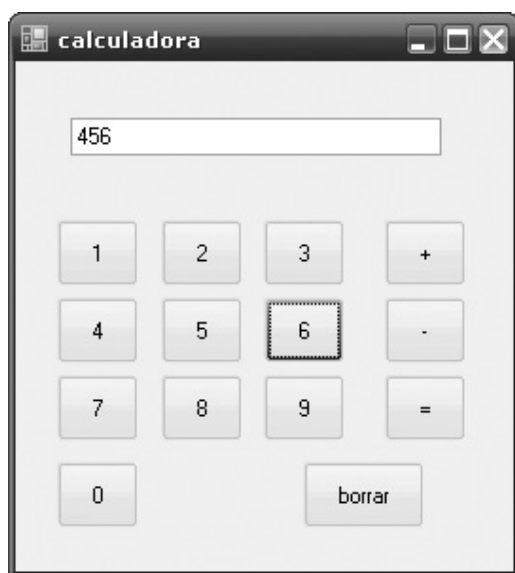


Figura 7.12 La calculadora.



Cuando se haga clic en el botón +, el número en pantalla deberá sumarse al total (haga lo mismo respecto del botón -, de manera que reste del total el número introducido). Al oprimir el botón = deberá aparecer el valor del total.

- 7.8 El elevador** Escriba un programa para simular un elevador muy primitivo. Para representarlo utilice un rectángulo relleno. Debe haber dos botones: uno para que suba por la pantalla y otro para que baje.
- 7.9 Nim** En este juego se utilizan fósforos o cerillos (no importa si son usados o nuevos, y tampoco cuántos), que se colocan en pilas (no interesa cuántos cerillos haya en cada pila). Los jugadores van por turnos; cada uno de ellos puede quitar cualquier cantidad de cerillos de cualquier pila, pero sólo de una. Además, cada jugador debe quitar por lo menos un cerillo. El ganador es quien hace que el otro jugador tome el último cerillo.

Escriba un programa para jugar este juego. Al principio la computadora debe repartir tres pilas de cerillos, con un número aleatorio (en el rango de 1 a 200) de cerillos en cada pila. La computadora debe participar como jugador: elegir una pila y una cantidad de cerillos al azar. El otro jugador será el usuario del programa, quien deberá especificar el número de pila y la cantidad de cerillos que tomará de ella mediante cuadros de texto, antes de hacer clic en un botón "Avanzar".

- 7.10 Gráficos de tortuga** Este tipo de gráficos facilita a los niños el aprendizaje de la programación. Imagine un bolígrafo fijado en la panza de una tortuga. A medida que ésta se desplaza por el piso, el bolígrafo va dejando un trazo. La tortuga puede recibir comandos de la siguiente manera:

- subir bolígrafo
- bajar bolígrafo
- girar bolígrafo 90° a la izquierda
- girar bolígrafo 90° a la derecha
- avanzar  $n$  píxeles

En un principio la tortuga se encuentra en las coordenadas 0, 0 y de cara a la derecha.

Con base en ello podemos dibujar un rectángulo si utilizamos la siguiente secuencia de ejemplo:

1. bajar bolígrafo
2. avanzar 20 píxeles
3. girar 90° a la derecha
4. avanzar 20 píxeles
5. girar 90° a la derecha
6. avanzar 20 píxeles
7. girar 90° a la derecha
8. avanzar 20 píxeles

Escriba un programa que se comporte como la tortuga, con un botón para cada uno de los comandos. Utilice una barra de seguimiento o un cuadro de texto para introducir el número de píxeles que debe avanzar ( $n$ ) la tortuga.

**SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**

**7.1** No, ya que consideran la edad específica de 18 años de manera distinta.

**7.2** La parte esencial de este programa es:

```
if (volumen < min)
{
    min = volumen;
}
label2.text = "El valor mínimo es " + Convert.ToString(min);
```

**7.3**

```
if (edad <= 16)
{
    label1.Text = "demasiado joven";
}
```

**7.4**

```
if ((total == 2) || (total == 5) || (total == 7))
{
    label2.Text = "usted gana";
}
```

**7.5**

```
if (edad >= 16 && edad < 65)
{
    MessageBox.Show("puede obtener un empleo");
}
```

**7.6**

```
int salario, impuestos;
salario = trackBar1.Value;
if ((salario > 10000) && (salario <= 50000))
{
    impuestos = (salario - 10000)/5;
}
if (salario > 50000)
{
    impuestos = 8000 + ((salario - 50000) * 9 / 10);
}
if (salario <= 10000)
{
    impuestos = 0;
}
```

**7.7**

```
private void button1_Click(object sender, EventArgs e)
{
    int a, b, c;
    int mayor;
```

```

    a = trackBar1.Value;
    b = trackBar2.Value;
    c = trackBar3.Value;
    if (a >= b && a >= c)
    {
        mayor = a;
    }
    else
    {
        if (b >= a && b >= c)
        {
            mayor = b;
        }
        else
        {
            mayor = c;
        }
    }
    MessageBox.Show("el mayor valor es " +
        Convert.ToString(mayor));
}

```

```

7.8 int edad;
    edad = Convert.ToInt32(textBox1.Text);
    if (edad >= 18 && edad <= 30)
    {
        textBox2.Text = "puede utilizar los servicios de la agencia";
    }

```

```

7.9 private string Convertir(int s)
    {
        string palo;
        switch(p)
        {
            case 1:
                palo = "diamantes";
                break;
            case 2:
                palo = "corazones";
                break;
            case 3:
                palo = "tréboles";
                break;

```

```
        case 4:
            palo = "picas";
            break;
        default:
            palo = "error";
            break;
    }
    return palo;
}
```



# Repetición

En este capítulo conoceremos cómo:

- realizar repeticiones mediante el uso de instrucciones `while`;
- realizar repeticiones mediante el uso de instrucciones `for`;
- utilizar los operadores lógicos `&&`, `||` y `!` en ciclos;
- realizar repeticiones mediante el uso de la instrucción `do`.

## ● Introducción

---

Los seres humanos estamos acostumbrados a realizar determinadas actividades —como comer, dormir y trabajar— una y otra vez. Las computadoras realizan repeticiones similares cuando, por ejemplo:

- suman una lista de números;
- buscan cierta información en un archivo;
- resuelven una ecuación matemática de manera iterativa, obteniendo cada vez mejores aproximaciones;
- hacen que una imagen gráfica se mueva en la pantalla (animación).

Hemos comentado ya que el trabajo de las computadoras consiste en obedecer *secuencias* de instrucciones. Ahora veremos cómo repetir una secuencia de instrucciones cierto número de veces. En parte, el poder de las computadoras deriva de su habilidad de realizar repeticiones con extrema rapidez. En el lenguaje de programación a las repeticiones se les conoce como *ciclos*, *bucles* o *loops*.

Son dos las principales formas en que los programadores en C# pueden instruir a la computadora para que realice una repetición: `while` y `for`. Es posible emplear cualquiera de ellas para llevar a cabo repeticiones pero, tal como veremos a continuación, su funcionamiento es diferente.

## ● while

---

Lo primero que haremos es utilizar un ciclo para mostrar los números enteros del 1 al 10 (Figura 8.1) en un cuadro de texto, para lo cual usaremos el siguiente código:

```
private void button1_Click(object sender, EventArgs e)
{
    int número;
    textBox1.Clear();
    número = 1;
    while (número <= 10)
    {
        textBox1.AppendText(Convert.ToString(número) + " ");
        número++;
    }
}
```

La palabra `while` indica que se requiere la repetición de las instrucciones que están encerradas entre las llaves (lo que se conoce como *cuerpo* del ciclo). La condición entre paréntesis que va inmediatamente después de la palabra `while` controla el ciclo. Si la condición se evalúa como verdadera, el ciclo continúa; de lo contrario se da por terminado y el control es transferido a la instrucción que está después de la llave de cierre. En este caso el ciclo continúa mientras cada `número` sea menor o igual a diez.

Antes de iniciar el ciclo, el valor de `número` se iguala a 1. Al final del ciclo el valor de `número` se incrementa una unidad mediante el uso del operador `++` que vimos en un capítulo anterior.

El cuadro de texto se vacía al inicio del programa mediante el método `clear`. En cada repetición del ciclo se agrega un número (y un espacio) al cuadro de texto, lo cual se logra con el método `AppendText`.

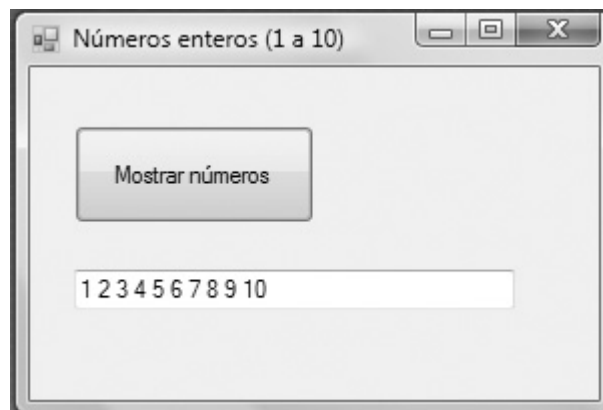


Figura 8.1 Despliegue de números enteros, del 1 al 10.

El fragmento del programa anterior utiliza el operador menor o igual que ( $\leq$ ). Éste es uno de varios operadores de comparación disponibles, mismos que utilizamos en las instrucciones `if`. He aquí nuevamente la lista completa de los operadores de comparación:

Símbolo	Significado
$>$	mayor que
$<$	menor que
$==$	igual que
$!=$	no es igual que
$\leq$	menor o igual que
$\geq$	mayor o igual que

La sangría, o indentación, que se aplica a las instrucciones que forman parte del ciclo (que por lo general el entorno de desarrollo de C# aplica de manera automática) nos ayuda a ver su estructura.

Una buena forma de comprender cómo funcionan los ciclos es mediante el uso del depurador, ya que nos permite seguir la ejecución del ciclo. En vez de hacer clic en el botón para iniciar la ejecución del programa, seleccione la opción **Paso a paso por instrucciones** del menú **Depurar**, u oprima la tecla de método abreviado correspondiente (en este caso, la tecla de función F11). Repita esta acción para avanzar paso a paso por el código del programa. Coloque el cursor sobre el texto **número** y observe cómo cambia su valor a medida que el ciclo se ejecuta. En el capítulo 9 veremos con más detalle el uso del depurador.

## PRÁCTICAS DE AUTOEVALUACIÓN

### 8.1 ¿Qué hace el siguiente fragmento de código?

```
int número = 0;
while (número <= 5)
{
    textBox1.AppendText(Convert.ToString(número * número)
                        + " ");
    número++;
}
```

### 8.2 Escriba un programa que sume (que calcule la suma de) los números del 1 al 100 y la muestre en un cuadro de texto al hacer clic en un botón.

El programa que analizaremos a continuación utiliza un ciclo `while` para desplegar una fila de cuadritos (Figura 8.2). El número de cuadritos se determina con base en el valor seleccionado en una barra de seguimiento. Cada vez que el apuntador de la barra cambia de posición se genera un evento y el programa muestra la figura con el número de cuadritos equivalente. Para lograrlo necesitamos un contador; éste, cuyo valor en un principio es igual a 1, se incrementa una unidad cada vez que se despliega un cuadrito. Es preciso repetir el proceso de mostrar un cuadrito adicional hasta que el contador llegue al total deseado, para lo cual utilizaremos un ciclo `while`, como se muestra a continuación:

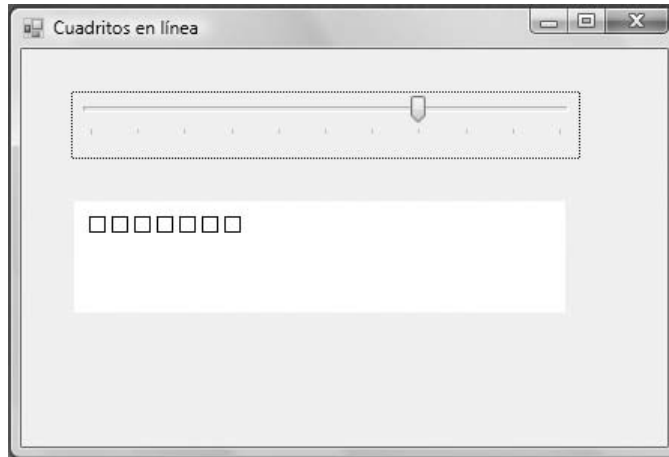


Figura 8.2 Uso del ciclo `while` para desplegar una fila de cuadritos.

```
private void trackBar1_Scroll(object sender, EventArgs e)
{
    int x, númeroDeCuadros, contador;
    Graphics papel;
    Pen lápiz = new Pen(Color.Black);

    númeroDeCuadros = trackBar1.Value;
    papel = pictureBox1.CreateGraphics();
    papel.Clear(Color.White);
    x = 10;
    contador = 1;
    while (contador <= númeroDeCuadros)
    {
        papel.DrawRectangle(lápiz, x, 10, 10, 10);
        x = x + 15;
        contador++;
    }
}
```

Este programa dibujará todos los cuadritos que usted quiera (por supuesto, esto dependerá de la longitud del cuadro de imagen y del valor máximo de la barra de seguimiento). Imagine cuántas instrucciones tendríamos que escribir para poder mostrar cien cuadros si no pudiéramos utilizar una instrucción `while`. Por otro lado, también hay que tener en cuenta que este programa dibujará cero cuadros si éste es el valor que el usuario selecciona con la barra de seguimiento. Como vemos, la instrucción `while` es completamente flexible: nos proporciona tantas repeticiones como se requiera.

Una forma de comprender cómo trabajan los ciclos `while` es por medio de un diagrama de acción como el que se muestra en la Figura 8.3. Por lo general la computadora lleva a cabo las instrucciones en secuencia de arriba hacia abajo, como indican las flechas. Los ciclos `while` implican que se debe



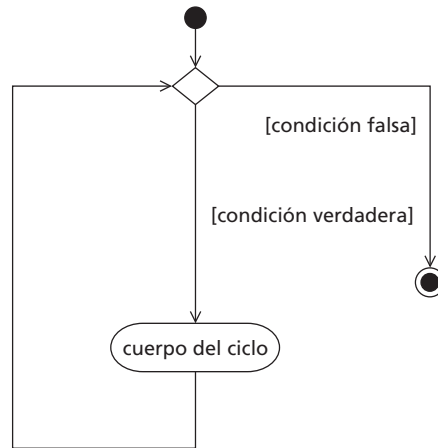


Figura 8.3 Diagrama de acción del ciclo `while`.

evaluar la condición antes de ejecutar el ciclo, y es necesario evaluar otra vez antes de que el ciclo se repita. Si la condición se evalúa como verdadera, se ejecuta el ciclo. Cuando llega el momento en que la condición se evalúa como falsa, el cuerpo del ciclo deja de ejecutarse y la repetición termina.

Es conveniente tener mucho cuidado al escribir ciclos `while` para asegurarnos de que el conteo se realice de manera apropiada. Un error común es hacer que el ciclo se repita una vez más o una vez menos de las necesarias. Esto se conoce comúnmente como error de “desplazamiento por uno” (*off by one*). Algunas veces se escriben ciclos que empiezan con un conteo de 0, y la intención de la evaluación es verificar si la condición es menor que el número requerido, como se muestra a continuación:

```

conteo = 0;
while (conteo < númeroRequerido)
{
    // cuerpo
    conteo = conteo++;
}

```

En otras ocasiones el ciclo comienza con un conteo de 1 y la evaluación busca corroborar si la condición es menor o igual al número requerido, como en el ejemplo siguiente:

```

conteo = 1;
while (conteo <= númeroRequerido)
{
    // cuerpo
    conteo = conteo++;
}

```

En este libro utilizaremos ambos estilos.

**PRÁCTICAS DE AUTOEVALUACIÓN**

**8.3 Rejas de la prisión** Escriba un programa que dibuje cinco líneas verticales paralelas.

**8.4 Tablero de ajedrez** Escriba un programa capaz de trazar un tablero de ajedrez a partir de nueve líneas verticales y nueve horizontales, con separación de 10 píxeles entre cada una de ellas.

**8.5 Números al cuadrado** Escriba un programa para mostrar los números del 1 al 5 y sus respectivos cuadrados (es decir, el resultado de elevar cada uno a la segunda potencia), un número (y su cuadrado) por línea. Use para ello el control `TextBox`, y modifique su propiedad `MultiLine` (líneas múltiples) de manera que especifique `True`. Use la cadena `"\r\n"` para avanzar al inicio de una nueva línea (`"\r"` hace que el cursor se desplace al inicio de una línea, y `"\n"` lo coloca en una nueva línea).

● **for**

En el ciclo `for` se agrupan muchos de los ingredientes de los ciclos `while` en el encabezado de la instrucción. Por ejemplo, veamos el programa anterior para mostrar los números 1 al 10, pero ahora utilizando `for`:

```
textBox1.Clear();
for (int número = 1; número <= 10; número++)
{
    textBox1.AppendText(Convert.ToString(número) + " ");
}
```

Dentro de los paréntesis de la instrucción `for` hay tres elementos separados por signos de punto y coma:

- una instrucción de inicialización: se lleva a cabo sólo una vez, antes de que inicie el ciclo. Ejemplo:

```
int número = 1
```

- una condición: se evalúa antes de cualquier ejecución del ciclo. Ejemplo:

```
número <= 10
```

- una instrucción final: ésta se lleva a cabo justo antes del final de cada ciclo. Ejemplo:

```
número++
```

La condición determina si el ciclo `for` se ejecuta o completa de cualquiera de las siguientes maneras:

- Si la condición es verdadera, se ejecuta el cuerpo del ciclo.
- Si la condición es falsa, el ciclo termina y se ejecutan las instrucciones que van después de la llave de cierre.

Tenga en cuenta que es posible escribir la declaración de una variable dentro de una instrucción `for` junto con su inicialización, algo que se hace con frecuencia. Esta variable puede utilizarse dentro del cuerpo de la instrucción `for`.

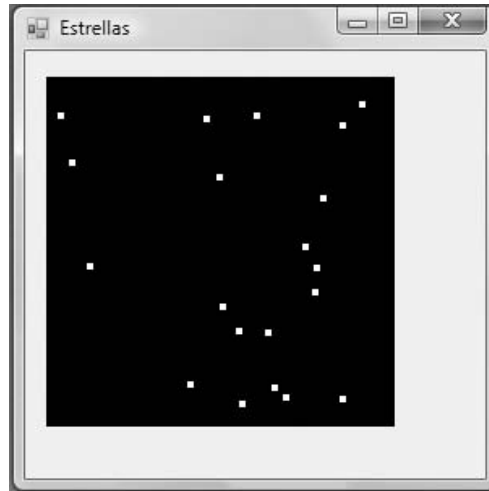


Figura 8.4 El programa Estrellas.

A continuación presentamos el código de otro programa de ejemplo que utiliza un ciclo `for` para mostrar veinte círculos pequeños en coordenadas aleatorias dentro de un cuadro de imagen de color negro (Figura 8.4).

```
private void pictureBox1_Click(object sender, EventArgs e)
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    papel.Clear(Color.Black);
    Random númeroAleatorio = new Random();
    SolidBrush miPincel = new SolidBrush(Color.White);

    for (int conteo = 0; conteo < 20; conteo++)
    {
        int x, y, radio;
        x = númeroAleatorio.Next(1, 200);
        y = númeroAleatorio.Next(1, 200);
        radio = 5;
        papel.FillEllipse(miPincel, x, y, radio, radio);
    }
}
```

Siempre es posible volver a codificar un ciclo `for` como un ciclo `while`, y viceversa; sin embargo, por lo general uno de los dos será más conveniente para una aplicación en particular.

#### PRÁCTICA DE AUTOEVALUACIÓN

**8.6** Vuelva a codificar el programa Estrellas, pero esta vez utilice `while` en vez de `for`.

### ● And, or, not

A veces la condición que controla la ejecución de los ciclos es más compleja, de manera que se hacen necesarios los operadores lógicos **and**, **or** y **not**. Es como si nos planteáramos algo así: “Daré un paseo hasta que empiece a llover, o hasta que sean las cinco de la tarde”. Comentamos ya estos operadores en el capítulo 7, en la sección sobre el uso de la instrucción **if** en las decisiones; se trata de:

Símbolo	Significado
&&	and (equivalente a la conjunción “y”)
	or (equivalente a la disyunción “o”)
!	not (equivale a la negación “no”)

Siguiendo con la analogía del mundo cotidiano, si quisiéramos describir mediante una instrucción **while** cuánto tiempo caminaremos diríamos: “Mientras no llueva y no sean las cinco de la tarde, seguiré caminando”. Observe que hay un “no” antes de cada una de las dos condiciones (lluvia, cinco de la tarde), y ambas están enlazadas por una “y”. Esto es lo que tiende a ocurrir cuando usamos ciclos **while**, de manera que debemos tener mucho cuidado y escribir la condición con toda claridad.

En el siguiente programa una persona ebria intenta alcanzar alguna de las paredes de una habitación cada vez que el usuario hace clic en un cuadro de imagen; la persona ebria da pasos de tamaño aleatorio (Figura 8.5). La posición que ocupa en cualquier instante se especifica mediante coordenadas *x* y *y*. Al principio la persona se halla en el centro del cuadro de imagen, y el ciclo continúa hasta que llega a una pared. En consecuencia, la instrucción **for** implica cuatro condiciones, enlazadas por operadores **&&**.



Figura 8.5 Caminata aleatoria.

```

private void pictureBox1_Click(object sender, EventArgs e)
{
    int x, y, pasoX, pasoY, nuevaX, nuevaY, pasos;
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    papel.Clear(Color.White);
    Random númeroAleatorio = new Random();
    Pen lápiz = new Pen(Color.Black);
    lápiz.Width = 4;
    x = pictureBox1.Width / 2;
    y = pictureBox1.Height / 2;
    for (pasos = 0;
        x < pictureBox1.Width && x > 0
        &&
        y < pictureBox1.Height && y > 0;
        pasos++)
    {
        pasoX = númeroAleatorio.Next(-50, 50);
        pasoY = númeroAleatorio.Next(-50, 50);
        nuevaX = x + pasoX;
        nuevaY = y + pasoY;
        papel.DrawLine(lápiz, x, y, nuevaX, nuevaY);
        x = nuevaX;
        y = nuevaY;
    }
    label1.Text = "Se necesitaron " + Convert.ToString(pasos)
        + " pasos para llegar a la pared";
}

```

## PRÁCTICA DE AUTOEVALUACIÓN

### 8.7 ¿Qué aparece en pantalla al ejecutarse el código siguiente?

```

int n, m;
n = 10;
m = 5;
while ((n > 0) || (m > 0))
{
    n = n - 1;
    m = m - 1;
}
MessageBox.Show("n = " + Convert.ToString(n) +
    " m = " + Convert.ToString(m));

```

### ● `do ... while`

---

Cuando se utilizan las instrucciones `while` o `for`, la evaluación se realiza siempre al principio de la repetición. El ciclo `do` es una estructura alternativa, en la cual la evaluación se lleva a cabo al final de cada repetición. Esto significa que el ciclo siempre se repite por lo menos una vez. Para ilustrar el uso del ciclo `do` escribiremos códigos para mostrar los números del 0 al 9 en un cuadro de texto mediante las tres estructuras de ciclo disponibles:

Si usamos `while`:

```
int conteo;
textBox1.Clear();
conteo = 0;
while (conteo <= 9)
{
    textBox1.AppendText(Convert.ToString(conteo) + " ");
    conteo++;
}
```

Si empleamos `for`:

```
textBox1.Clear();
for (int conteo = 0; conteo <=9; conteo++)
{
    textBox1.AppendText(Convert.ToString(conteo) + " ");
}
```

Si usamos `do` (la evaluación se hace al final del ciclo):

```
int conteo;
textBox1.Clear();
conteo = 0;
do
{
    textBox1.AppendText(Convert.ToString(conteo) + " ");
    conteo++;
}
while (conteo < 10);
```

Veamos ahora un ejemplo que exige la ejecución de un ciclo por lo menos una vez. En los juegos de lotería los números se seleccionan al azar, pero no puede haber dos números iguales. Consideremos una lotería muy pequeña, en la que sólo se seleccionan dos números. Tras obtener el primer número aleatorio necesitamos un segundo número, pero éste no debe ser igual al primero. En consecuencia, seleccionamos números tantas veces como sea necesario para que sea distinto del primero. He aquí el código resultante de este razonamiento:

```
private void button1_Click(object sender, EventArgs e)
{
    Random númeroAleatorio = new Random();
    int número1, número2;

    número1 = númeroAleatorio.Next(1, 10);
    do
    {
        número2 = númeroAleatorio.Next(1, 10);
    }
    while (número1 == número2);
    label1.Text = "los números son "
        + Convert.ToString(número1) + " y "
        + Convert.ToString(número2);
}
```

## ● Ciclos anidados

Los ciclos anidados son ciclos dentro de otros ciclos. Suponga, por ejemplo, que deseamos mostrar la pantalla de la Figura 8.6, la representación rudimentaria de un bloque de apartamentos. Imagine que el bloque consta de cuatro pisos, cada uno con cinco apartamentos, los cuales aparecen como rectángulos. El ciclo para dibujar un piso individual tiene la siguiente estructura:

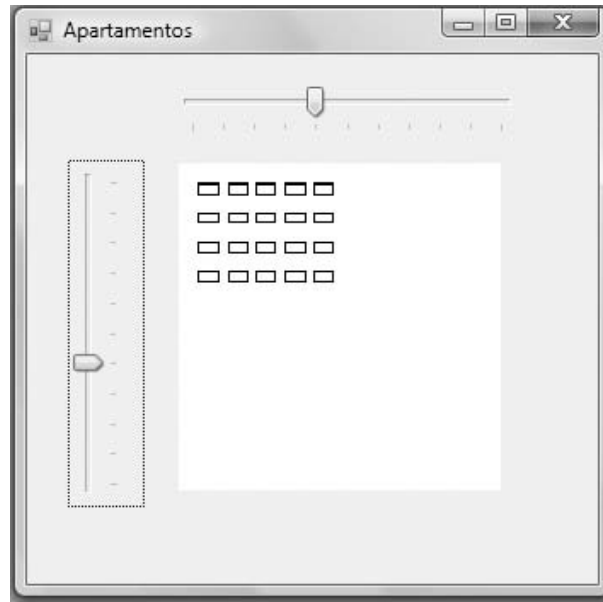


Figura 8.6 Representación de un bloque de apartamentos.

```

for (int apartamento = 1; apartamento <= 5; apartamento++)
{
    // código para dibujar un apartamento
}

```

y el ciclo para dibujar varios pisos tiene la siguiente estructura:

```

for (int piso = 1; piso <= 3; piso++)
{
    // código para dibujar un piso
}

```

Lo que necesitamos es encerrar el primer ciclo dentro del segundo, de manera que los ciclos estén anidados.

Podemos usar barras de seguimiento para establecer la cantidad de apartamentos por piso y el número de pisos del bloque. Cada vez que manipulemos cualquiera de las barras de seguimiento se producirá un evento y se invocará este método:

```

private void DibujarApartamentos(int pisos, int apartamentos)
{
    int x, y;
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    papel.Clear(Color.White);
    Pen miLápiz = new Pen(Color.Black);
    y = 10;
    for (int piso = 0; piso <= pisos; piso++)
    {
        x = 10;
        for (int conteo = 0; conteo <= apartamentos; conteo++)
        {
            papel.DrawRectangle(miLápiz, x, y, 10, 5);
            x = x + 15;
        }
        y = y + 15;
    }
}

```

En el ejemplo anterior podemos ver que la sangría nos ayuda de manera considerable a comprender el funcionamiento del programa. Por supuesto, siempre tenemos la opción de rediseñar los ciclos anidados mediante el uso de métodos, y algunas veces esto resulta más claro. En el capítulo 21 exploraremos dicha alternativa con más detalle, cuando hablemos sobre los estilos de programación.

## PRÁCTICA DE AUTOEVALUACIÓN

**8.8** Las partituras de música se escriben en papel impreso con pentagramas. Cada pentagrama consiste en cinco líneas horizontales impresas a lo largo de la página, con espacios de aproximadamente 2 mm (1/10 de pulgada) entre ellas. Cada página contiene ocho de estos pentagramas. Escriba un programa para dibujar una página para partitura musical.



## ● Combinación de las estructuras de control

En el capítulo anterior vimos cómo implementar la selección mediante las instrucciones `if` y `switch`, y en éste analizamos la repetición mediante el uso de las instrucciones `while`, `for` y `do`. Casi todos los programas utilizan combinaciones de estas estructuras de control. De hecho, la mayoría de ellos consisten de:

- secuencias;
- ciclos;
- selecciones;
- llamadas a métodos de biblioteca;
- llamadas a métodos que nosotros, los programadores, escribimos.

### Fundamentos de programación

Las repeticiones (ciclos o bucles) se utilizan mucho en programación. En C# hay tres variedades de instrucciones para dicho propósito: `while`, `for` y `do`. ¿Cuál debemos usar? Las instrucciones `while` y `for` son similares, pero esta última suele usarse cuando hay un contador asociado al ciclo. Por ende, el prototipo del ciclo `for` tiene la siguiente estructura:

```
for ( int conteo = 0; conteo <= valorFinal; conteo++ )
{
    // cuerpo del ciclo
}
```

El contador `conteo` tiene un valor inicial, un valor final y se incrementa cada vez que se repite el ciclo. El ciclo `while` se utiliza cuando no puede calcularse por anticipado el número de repeticiones de que constará el ciclo; de esta manera el ciclo continúa mientras que la condición sea verdadera. Un ejemplo es el programa de caminata aleatoria que vimos antes, en donde la repetición continúa hasta que la persona alcanza una pared de la habitación. El prototipo del ciclo `while` tiene esta estructura:

```
while (condición)
{
    // cuerpo del ciclo
}
```

El ciclo `do` se utiliza cuando es necesario que la evaluación que termine la repetición se haga al final del ciclo; recuerde que los ciclos `do` siempre se llevan a cabo por lo menos una vez.

Los ciclos demuestran su valor en los programas que procesan colecciones de datos. Más adelante veremos varias colecciones, incluyendo cuadros de lista, cadenas de caracteres, archivos y arreglos.

### Errores comunes de programación

- Siempre que se escriben ciclos hay que tener mucho cuidado al especificar la condición. Un error muy común consiste en hacer que un ciclo termine una repetición antes o que se repita una vez de más. A esto se le conoce algunas veces como error de “desplazamiento por uno” (*off by one*).

- Trate de evitar la introducción de condiciones complejas en los ciclos. Por ejemplo, ¿necesita `||` o `&&`?
- Si el cuerpo de un ciclo consta de una sola instrucción no necesita estar rodeado de llaves; sin embargo, por lo general es más seguro utilizarlas, y ésta es la metodología que utilizaremos en este libro.

## Secretos de codificación

- El ciclo `while` tiene la siguiente estructura:

```
while (condición)
{
    instrucción(es)
}
```

en donde la condición se evalúa antes de cualquier repetición del ciclo. Si es verdadera, el ciclo continúa; si es falsa, el ciclo termina.

- El ciclo `for` tiene esta estructura:

```
for (acción inicial; condición; acción)
{
    instrucción(es)
}
```

en donde:

`acción inicial` se lleva a cabo una vez, antes de ejecutar el ciclo.

`condición` se evalúa antes de cada repetición. Si es verdadera se repite el ciclo; de lo contrario el ciclo termina.

`acción` se lleva a cabo al final de cada repetición.

- El ciclo `do` tiene la siguiente estructura:

```
do
{
    instrucción(es)
}
while (condición)
```

En este caso la evaluación se realiza después de cada repetición.

## Nuevos elementos del lenguaje

Las estructuras de control para repetición:

```
while
for
do
```

## Resumen

- En programación a las repeticiones se les conoce como *ciclos*, *bucles* o *loops*.
- En C# hay tres instrucciones para indicar a la computadora que realice un ciclo: `while`, `for` y `do`.
- La instrucción `for` se utiliza cuando queremos describir las características principales del ciclo dentro de la instrucción del mismo.
- La instrucción `do` se utiliza cuando debe evaluarse una condición al final de un ciclo, y/o cuando el ciclo debe realizarse por lo menos una vez.

## EJERCICIOS

- 8.1 Despliegue de números enteros** Escriba un programa que utilice un ciclo para mostrar los números enteros del 1 al 10, junto con sus cubos (es decir, el resultado de elevar cada uno a la tercera potencia).
- 8.2 Números aleatorios** Escriba un programa para mostrar diez números aleatorios utilizando un ciclo. Use la clase `Random` para obtener números aleatorios en el rango de 1 a 100. Haga que los números se desplieguen en un cuadro de texto.
- 8.3 La vía láctea** Escriba un programa que dibuje cien círculos en un cuadro de imagen, de manera que ocupen posiciones aleatorias y cuenten con diámetros al azar, de hasta 100 píxeles.
- 8.4 Escalones** Escriba un programa para trazar una serie de escalones hechos de ladrillos, como se muestra en la Figura 8.7. Use el método de biblioteca `DrawRectangle` para dibujar cada ladrillo.
- 8.5 Suma de enteros** Escriba un programa que sume los números del 0 al 39 utilizando un ciclo. Compruebe que las respuestas obtenidas sean correctas; emplee para ello la fórmula para la suma de los números de 0 a  $n$ :

$$\text{suma} = n \times (n + 1)/2$$



Figura 8.7 Escalones.

- 8.6 Patrón de diente de sierra** Escriba un programa para mostrar un patrón de diente de sierra en un cuadro de texto, como se muestra en la Figura 8.8. El programa deberá utilizar la cadena "\t\n" para obtener una nueva línea.

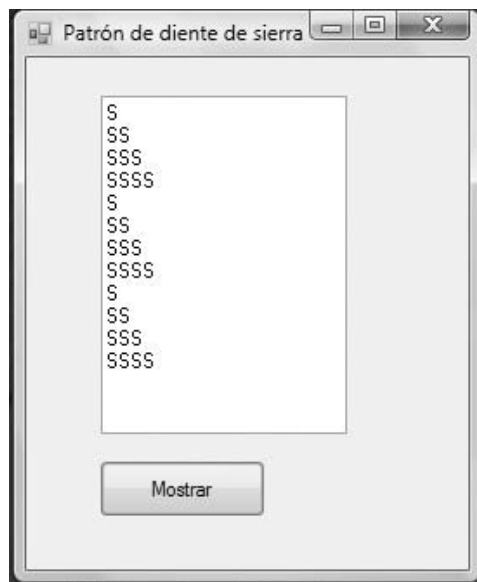


Figura 8.8 Patrón de diente de sierra.

- 8.7 Tabla de multiplicar** Escriba un programa para mostrar una tabla de multiplicación, como la que utilizan los niños pequeños. Por ejemplo, la tabla para los números hasta el 5 se muestra en la Figura 8.9. Además de utilizar la cadena "\t\n" para obtener una nueva línea, el programa debe usar la cadena "\t" para avanzar el tabulador a la siguiente posición, de manera que la información se muestre en columnas ordenadas.

Además, el programa debe ser capaz de mostrar una tabla de cualquier tamaño, el cual se debe especificar introduciendo un entero en un cuadro de texto.

- 8.8 Fibonacci** La serie de Fibonacci es la serie de números que se muestra a continuación:

1 1 2 3 5 8 13 ...

Cada número (excepto los primeros dos) es la suma de los dos números anteriores. Los primeros dos números son 1 y 1. Se supone que esta serie gobierna el crecimiento en las plantas. Escriba un programa para calcular y mostrar los primeros 20 números de Fibonacci.

- 8.9 Suma de series** Escriba un programa para calcular y mostrar la suma de las series:

$1 - 1/2 + 1/3 - 1/4 + \dots$

hasta llegar a un término que sea menor a 0.0001.

- 8.10 Canción de cuna** Escriba un programa para mostrar todos los versos de una canción de cuna en un cuadro de texto con una barra de desplazamiento vertical. La primera estrofa es:

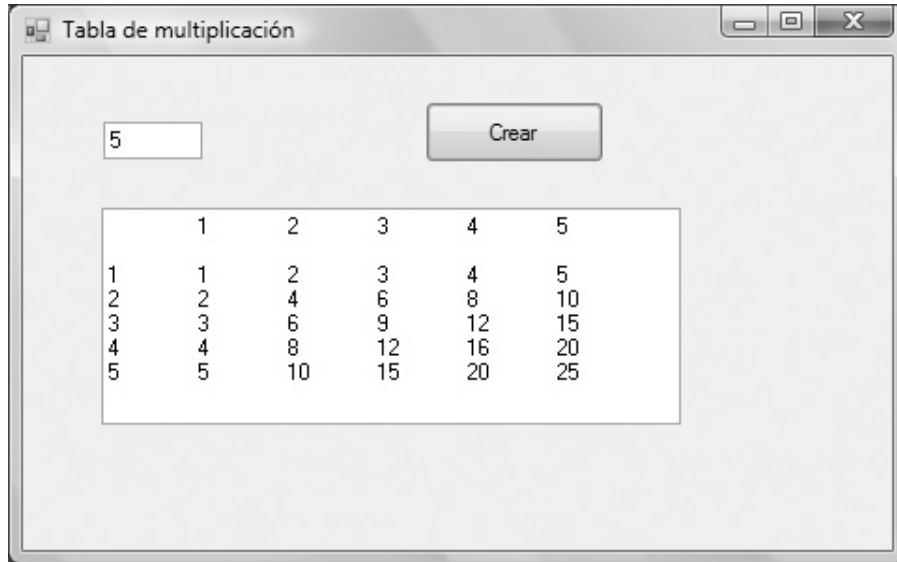


Figura 8.9 Tablas de multiplicación.

*10 botellas verdes, colgando de la pared,  
 10 botellas verdes, colgando de la pared,  
 Si a 1 botella verde se le ocurre caer  
 habrá 9 botellas verdes, colgando de la pared*

En los versos subsiguientes hay cada vez menos botellas, a medida que se “caen” de la pared.

## SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

**8.1** Despliega los números 0 1 4 9 16 25.

**8.2**

```
private void button1_Click(object sender, EventArgs e)
{
    int número;
    int suma;

    suma = 0;
    número = 1;
    while (número <= 100)
    {
        suma = suma + número;
        número++;
    }
    textBox1.Text = "La suma es " + Convert.ToString(suma);
}
```

Este programa utiliza una técnica de programación común: un total acumulado. Al principio el valor de la variable `suma` es igual a cero. Cada vez que se repite el ciclo, el valor de `número` se suma al valor de `suma`, y el resultado se coloca en la variable `suma` una vez más.

```
8.3    private void pictureBox1_Click(object sender, EventArgs e)
        {
            int x, contador, númeroDeRejas;
            Graphics papel;
            Pen miLápiz = new Pen(Color.Black);

            númeroDeRejas = 5;
            papel = pictureBox1.CreateGraphics();
            papel.Clear(Color.White);
            x = 10;
            contador = 1;
            while (contador <= númeroDeRejas)
            {
                papel.DrawLine(miLápiz, x, 10, x, 100);
                x = x + 15;
                contador++;
            }
        }
```

```
8.4    private void pictureBox1_Click(object sender, EventArgs e)
        {
            int x, y, contador;
            Graphics papel;
            Pen miLápiz = new Pen(Color.Black);

            papel = pictureBox1.CreateGraphics();
            papel.Clear(Color.White);

            x = 10;
            contador = 1;
            while (contador <= 9)
            {
                papel.DrawLine(miLápiz, x, 10, x, 90);
                x = x + 10;
                contador++;
            }

            y = 10;
            contador = 1;
            while (contador <= 9)
```

```

        {
            papel.DrawLine(miLápiz, 10, y, 90, y);
            y = y + 10;
            contador++;
        }
    }

```

**8.5** `private void button1_Click(object sender, EventArgs e)`

```

{
    int número = 1;
    while (número <= 5)
    {
        textBox1.AppendText(Convert.ToString(número) + " " +
            Convert.ToString(número * número) + "\r\n");
        número++;
    }
}

```

**8.6** La estructura fundamental del ciclo es:

```

int conteo = 0;
while (conteo < 20)
{
    // cuerpo del ciclo
    conteo++;
}

```

**8.7** `n = 0` y `m = -5`.

**8.8** `private void pictureBox1_Click(object sender, EventArgs e)`

```

{
    int y;
    Pen miLápiz = new Pen(Color.Black);
    Graphics papel = pictureBox1.CreateGraphics();
    papel.Clear(Color.White);
    y = 10;
    for (int pentagramas = 1; pentagramas <= 8; pentagramas++)
    {
        for (int líneas = 1; líneas <= 5; líneas++)
        {
            papel.DrawLine(miLápiz, 10, y, 90, y);
            y = y + 2;
        }
        y = y + 5;
    }
}

```



# Depuración

## En este capítulo conoceremos:

- los distintos tipos de errores de programación (*bugs*);
- cómo utilizar el depurador;
- cómo utilizar los puntos de interrupción y el paso a paso por instrucciones;
- los errores de programación más comunes.

## ● Introducción

Depuración es el nombre que se da a la tarea de averiguar si un programa incluye errores de codificación y en dónde se encuentran éstos, para después poder corregir el problema. El término que se da en inglés a esta acción es *debugging*, una derivación de la palabra *bug*, que significa “insecto”, “bicho”, y que en el contexto de programación se refiere a un error de codificación. Esta acepción se originó en los días de las computadoras de válvulas, cuando (según cuenta la historia) un enorme insecto quedó atrapado en los circuitos de una de las primeras computadoras, provocando que fallara.

Ahora bien, debido a que los programadores somos seres humanos imperfectos, todos los programas tienden a contar con este tipo de errores... en especial la primera vez que los escribimos. Por fortuna, el IDE de C# proporciona un depurador para ayudarnos a encontrarlos.

El depurador de C# también es una herramienta que nos permite comprender cómo funcionan las variables, las asignaciones, las instrucciones `if` y los ciclos, ya que (como veremos) podemos utilizarlo para visualizar la ejecución de nuestros programas:

Como parte de la historia sobre el origen de los errores, rastreamos lo que sucede cuando corre un programa. Esta acción incluye tres etapas:

1. compilación;
2. vinculación;
3. ejecución.

A continuación analizaremos cada una de estas fases por separado.



## Compilación

A medida que escribimos un programa, el entorno de desarrollo de C# lleva a cabo comprobaciones exhaustivas para exponer numerosos errores que de otra forma podrían persistir. Dichas equivocaciones aparecen subrayadas tan pronto como escribimos las instrucciones del programa, y se denominan errores de compilación. Los ejemplos más comunes son una variable no declarada, o la omisión de un punto y coma o un paréntesis.

Una vez corregidos todos los errores, el programa se compilará “limpiamente”, y se ejecutará aun cuando tal vez no haga exactamente lo que queríamos.

## Vinculación

Todos los programas utilizan métodos de biblioteca, y algunos emplean también clases escritas por el programador. Estas clases sólo se vinculan cuando el programa empieza a ejecutarse. No obstante, como mencionamos antes, el IDE comprueba el programa tan pronto como se va escribiendo el código correspondiente, para garantizar que todos los métodos invocados existan y que los parámetros coincidan en número y tipo. Insistimos: todos los errores se muestran subrayados al momento de escribir las instrucciones del programa.

## Ejecución

Aunque el programa llegue a la fase de ejecución es muy raro que la primera vez funcione como se desea. De hecho es común que falle de cierta manera, o que se comporte de forma distinta a la esperada. Algunos errores (como los intentos de dividir cantidades `int` entre cero) se detectan de manera automática, tras lo cual el programador recibe una notificación; otros son más sutiles y simplemente producen un comportamiento inesperado. No importa si tiene uno o muchos errores en su programa; debe llevar a cabo un proceso de depuración.

Más adelante en este capítulo veremos ejemplos de errores comunes que surgen al programar en C#. Por ahora, lo importante es entender que la depuración implica un problema: los síntomas que indican la presencia de un error por lo general resultan muy poco informativos. Por lo tanto, tenemos que recurrir al trabajo detectivesco para determinar la causa. Es como ser médicos: ante la presencia de síntomas tenemos que encontrar la causa, y después corregir el problema.

Una vez que eliminemos las fallas más obvias de un programa, por lo común empezaremos a realizar pruebas sistemáticas. Estas comprobaciones consisten en la ejecución repetida del programa con una variedad de datos como entrada; hablaremos sobre este proceso en el capítulo 22. El objetivo de las pruebas es convencer al mundo de que el programa funciona de manera apropiada, pero por lo general revelan la existencia de más errores. Entonces es necesario depurar más el programa; de aquí que los procesos de prueba y depuración vayan de la mano.

A muchos programadores les gusta el proceso de depuración; lo encuentran emocionante, algo así como ver una película de misterio en la que el villano se revela hasta el último momento. Sin duda el proceso de depuración (junto con las pruebas) requiere bastante tiempo; no se preocupe por cuánto tarde, ¡es normal!

## ● Uso del depurador

Imaginemos que un programa se ejecuta pero se comporta de manera inesperada. ¿Cómo podemos averiguar la causa del problema? Casi todos los programas muestran algo en pantalla, pero en realidad podríamos decir que su trabajo es invisible. Para apreciarlo y ver cómo se está comportando el programa necesitamos una especie de lentes de rayos X. Ésa es la clave para una depuración exitosa: obtener información adicional sobre el programa en ejecución.

El entorno de desarrollo integrado de C# proporciona un *depurador*. Se trata de un programa que se ejecuta junto al nuestro, permitiéndonos inspeccionar su progreso. El depurador cuenta con varias herramientas, incluyendo el recorrido paso a paso por las instrucciones y los puntos de interrupción.

### Puntos de interrupción

El depurador permite que el programador coloque *puntos de interrupción* en el programa. Los puntos de interrupción son lugares específicos en los que se detiene la ejecución del programa, y se insertan de la siguiente manera:

1. Haga clic en la barra de color gris que está a la izquierda de cualquiera de las líneas de codificación del programa (Figura 9.1). La línea quedará resaltada en color marrón, y aparecerá un círculo del mismo color sobre la barra gris.
2. Inicie el programa en la forma usual.

Al llegar al punto de interrupción el programa se detendrá justo antes de ejecutar la línea, y el resaltado de la misma cambiará de color marrón a amarillo; además aparecerá una flecha amarilla sobre el círculo. En este momento usted puede colocar el cursor sobre el nombre de una variable o una propiedad, y ver su valor desplegado en un cuadro emergente especial. Cualquier discrepancia entre el valor real y el valor que debería tener nos proporciona valiosa información para la depuración.

En vez de ver los valores al colocar el cursor sobre el nombre de una variable o propiedad, podemos crear una ventana de inspección independiente, que muestra los valores de variables seleccionadas. Para ello haga clic con el botón derecho del ratón sobre el nombre de una variable; enseguida aparecerá un menú contextual como el que se ilustra en la Figura 9.2.

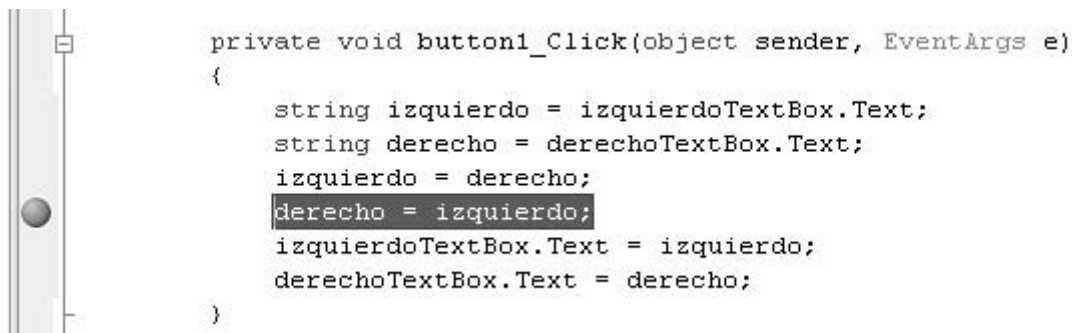


Figura 9.1 Creación de un punto de interrupción.

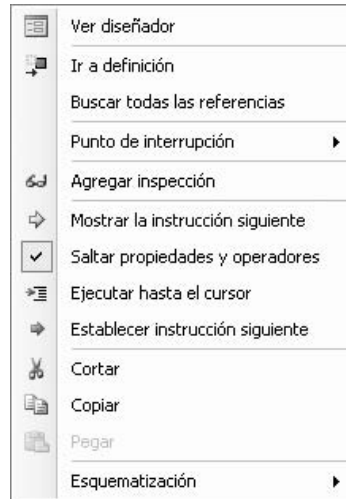


Figura 9.2 El menú que aparece al hacer clic con el botón derecho del ratón en el nombre de una variable.



Figura 9.3 Una ventana de inspección.

Al seleccionar **Agregar inspección** se crea una ventana de inspección en la parte inferior de la pantalla, con el nombre de la variable y su valor. En la Figura 9.3 se muestra un ejemplo de dicha ventana, en la cual aparecen dos variables; no obstante, es posible agregar cualquier número de variables a ella, de manera que podamos vigilar sus valores constantemente.

Una vez que obtengamos la información deseada, podremos remover un punto de interrupción individual con sólo hacer clic en el punto marrón que está a la izquierda de la línea de código correspondiente.

Es posible insertar varios puntos de interrupción en un programa. En este caso, una vez que lleguemos a un punto de interrupción podemos hacer clic en el botón **Continuar** de la barra de herramientas **Depurar** para reanudar la ejecución del programa hasta que termine, o hasta llegar al siguiente punto de interrupción. Para borrar un punto de interrupción haga clic sobre él; también puede hacer clic con el botón derecho del ratón y seleccionar la opción **Eliminar punto de interrupción** en el menú contextual.

El truco para encontrar errores con rapidez consiste en elegir los mejores lugares del código en donde podamos colocar puntos de interrupción. Por lo general, éstos son:

- al principio de un método, para comprobar que los parámetros sean correctos;
- justo después de la invocación a un método (para verificar que éste haya realizado correctamente su trabajo), o justo al final del método (para cumplir el mismo propósito).

## Revisión paso a paso del programa

El depurador también nos permite ejecutar los programas una línea a la vez; a esto se le conoce como *recorrido paso a paso*. Para ello debemos seleccionar el comando **Paso a paso por instrucciones** del menú **Depurar**, aunque es más conveniente usar una tecla de función (el número de esa tecla de función depende de la configuración de su equipo; abra el menú **Depurar**, mismo que se muestra en la Figura 9.4, para averiguar cuál es). El programa ejecutará una línea y se detendrá antes de aquella que esté resaltada. Cualquier diferencia entre la ruta de ejecución esperada y la real nos proporciona información útil sobre la causa del error. Además, en cualquier momento del proceso podemos colocar el cursor sobre el nombre de una variable para observar su valor, o abrir una ventana de inspección como vimos antes. Una vez más: cualquier diferencia entre el valor esperado y el valor actual nos proporciona datos valiosos para nuestra depuración.

Aunque podemos llegar a divertirnos mucho con un depurador, la desventaja es que su labor puede requerir mucho tiempo. A continuación le presentamos una manera productiva de utilizarlo:

1. Haga una hipótesis sobre la posible ubicación del error con base en los problemas que presente su programa. Tal vez de esta manera logre predecir que el error se encuentra dentro de dos o tres métodos cualesquiera.
2. Coloque puntos de interrupción a la entrada y salida de los métodos sospechosos.
3. Ejecute el programa. Cuando éste se detenga en la entrada de un método, inspeccione los valores de los parámetros. A la salida inspeccione el valor de retorno y los valores de las variables de instancia importantes. Con esto podrá identificar el método dentro del cual está el error.
4. Ejecute el programa de nuevo y deténgase en la entrada del método con el error. Avance paso a paso por él hasta que detecte la discrepancia entre lo esperado y la realidad; entonces habrá encontrado el error.

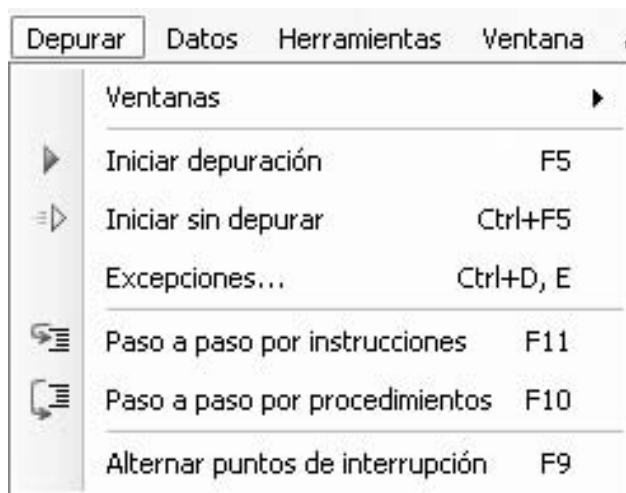


Figura 9.4 Las opciones del menú Depurar.

## ● Ejemplo práctico de depuración

El programa que revisaremos a continuación pretende servir como una declaración de amor entre dos enamorados (en el caso del ejemplo de la Figura 9.5, Romeo y Julieta). Hay un botón para intercambiar los nombres que se introduzcan en los dos cuadros de texto. Sin embargo, tal como está escrito el programa no hace la modificación esperada. El código erróneo es:

```
private void button1_Click(object sender, EventArgs e)
{
    string izquierdo = izquierdoTextBox.Text;
    string derecho = derechoTextBox.Text;
    izquierdo = derecho;
    derecho = izquierdo;
    izquierdoTextBox.Text = izquierdo;
    derechoTextBox.Text = derecho;
}
```

Sólo hay un método en este programa, por lo que el origen del problema debe estar en él. Para detectarlo colocamos un punto de interrupción al principio del método y avanzamos paso a paso por las instrucciones, colocando el cursor sobre las dos variables (los valores de los cuadros de texto) para ver cuáles son sus parámetros. También podemos crear una ventana de inspección, como vimos antes. De inmediato se deduce que el valor de `izquierdo` se sobrescribió y, en consecuencia, se perdió. Hemos cometido un error clásico, que podemos rectificar si almacenamos este valor en una variable temporal, como en el código siguiente:

```
private void button1_Click(object sender, EventArgs e)
{
    string temp;
    temp = izquierdoTextBox.Text;
    izquierdoTextBox.Text = derechoTextBox.Text;
    derechoTextBox.Text = temp;
}
```

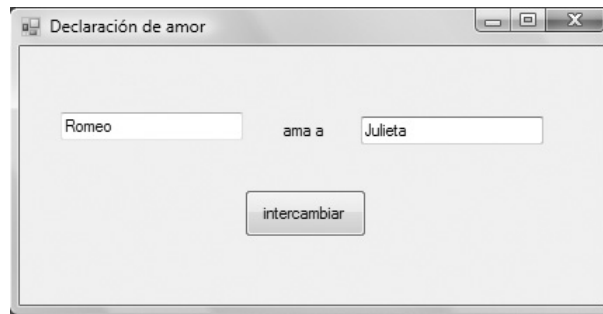


Figura 9.5 El programa Declaración de amor.

## ● Errores comunes

---

Los programadores en C# comúnmente cometen ciertos errores. A continuación listaremos algunos de ellos. Vale la pena comprobar si cualquier programa sospechoso los incluye.

### Errores de compilación

El entorno de desarrollo de C# lleva a cabo muchas comprobaciones de nuestros programas a medida que escribimos el código, y muestra una línea de color rojo debajo de cualquier instrucción que interprete como potencialmente problemática. A estos problemas se les denomina errores de compilación o de sintaxis, y es posible obtener una breve explicación del error si colocamos el cursor sobre el texto señalado. El propósito de esta característica es tratar de garantizar que los programas creados con C# sean robustos. Cualquier error que se detecte en tiempo de compilación puede corregirse con facilidad, pero aquellos que permanecen ignorados hasta el tiempo de ejecución podrían exigir mucha depuración. Por lo tanto, aunque la detección de errores en tiempo de compilación puede resultar molesta, es muy valiosa.

La fuente más común de errores en los programas de C# es la omisión de signos de punto y coma y paréntesis. Por fortuna el IDE de C# es muy bueno para detectar estos errores e incluso sugiere la manera de corregirlos.

He aquí algunas otras áreas que frecuentemente producen errores de compilación:

#### ***Nombres de variables***

Todas las variables deben declararse, y es preciso que a partir de ese momento sus nombres se escriban de la misma manera en forma consistente. En ocasiones es tentador usar una palabra clave como nombre de una variable (por ejemplo, la palabra **event** es una palabra clave).

#### ***Nombres de métodos y propiedades***

Es muy común:

- omitir una instrucción **using**;
- escribir mal el nombre de un método o de una propiedad;
- obtener los parámetros incorrectos para un método.

#### ***Conversiones***

Si usted escribe:

```
textBox1.Text = 23;
```

recibirá un mensaje de error (es decir, el código aparecerá subrayado con una línea de color rojo) para indicarle que no se permite una conversión implícita de número a cadena de texto. La conversión debe realizarse de la siguiente forma:

```
textBox1.Text = Convert.ToString(23);
```

Un error similar podría ocurrir al introducir un número mediante un cuadro de texto:

```
int número  
número = textBox1.Text;
```

Esto produce un mensaje de error de compilación, en el cual se indica que el texto tiene que ser convertido de manera explícita en un número. Para remediar este problema podemos realizar la conversión de la siguiente forma:

```
número = Convert.ToInt32(textBox1.Text);
```

### Errores en tiempo de ejecución

Los errores en tiempo de ejecución ocurren a medida que se ejecuta el programa, y son detectados por el sistema en ese momento. De nuevo, esto forma parte de las medidas diseñadas para asegurar que los programas sean robustos o, en otras palabras, para evitar que un programa con errores actúe como un toro suelto en una tienda de porcelana china, con la probabilidad de que la computadora falle. Los errores en tiempo de ejecución hacen que aparezca un mensaje de error y que se detenga el programa.

#### Excepciones aritméticas

Si un programa intenta dividir entre cero, se detendrá y aparecerá un mensaje de error. Es muy fácil dejar que esto ocurra de manera inadvertida, por ejemplo en un programa que contenga este fragmento de código:

```
int a, b, c;  
b = 1;  
c = Convert.ToInt32(textBox1.Text);  
a = b / c;
```

Si el usuario introduce el número cero en el cuadro de texto, el programa intentará realizar una división entre cero, se detendrá y desplegará un cuadro de mensaje (Figura 9.6). La línea de C# que produjo el error aparecerá resaltada en color amarillo.

El mensaje nos indica con bastante claridad lo que ha ocurrido.

Un remedio para esta situación consiste en escribir código para comprobar el valor de *c* antes de realizar la división.

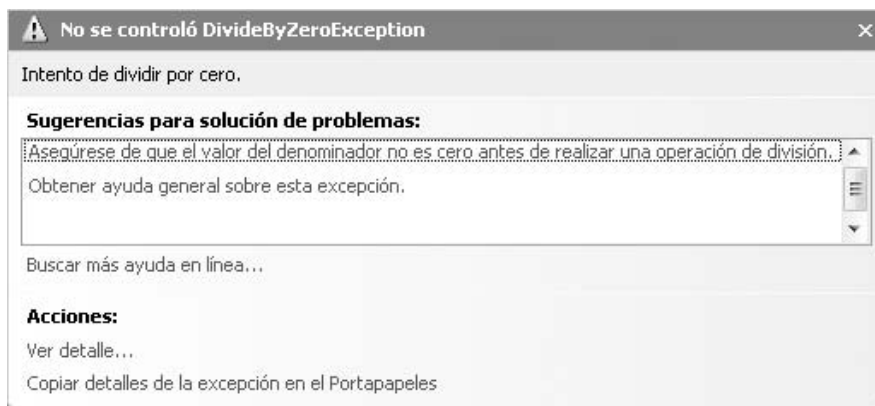


Figura 9.6 División entre cero.

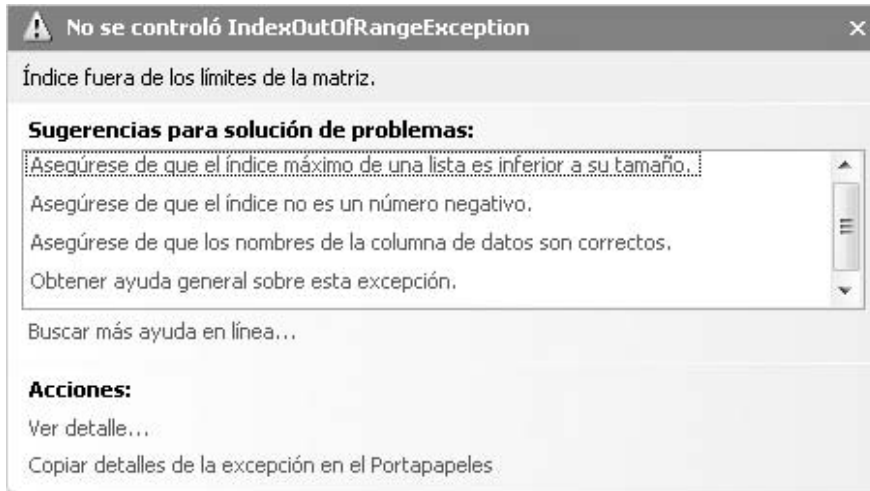


Figura 9.7 Índice de arreglo fuera de rango.

### Índices de arreglos

Los arreglos (o matrices) constituyen un tema que explicaremos hasta el capítulo 14. Sin embargo y por cuestión de integridad, incluiremos en esta sección la mención de un error común que ocurre al utilizar arreglos. Si declaramos un arreglo de la siguiente forma:

```
int[] tabla = new int[ 10 ];

for ( int índice = 0; índice <= 10; índice++ ) // advertencia, es incorrecto
{
    tabla[ índice ] = 0;
}
```

el código pondrá un cero en cada uno de los diez elementos del arreglo `tabla`, pero luego tratará de colocar un cero en el valor de índice 10, un ejemplo clásico del error de desplazamiento por uno (*off by one*). Este valor se encuentra más allá del final del arreglo, por lo que el programa falla y aparece un mensaje de error (Figura 9.7). La instrucción de C# que produjo el problema se muestra resaltada. El mensaje es bastante claro, y nos indica que el índice está fuera de los límites.

La solución de este problema es corregir la falla de diseño en el código.

### Uso de un objeto inexistente

En capítulos anteriores vimos cómo declarar un objeto como instancia de una clase. Por ejemplo, podemos declarar una variable `adivinatorEdad` como una instancia de la clase `Random`:

```
Random adivinatorEdad;
```

Si ahora tratamos de usar el objeto `adivinatorEdad` mediante una invocación a su método `Next` de la siguiente forma::

```
int edad = adivinatorEdad.Next[5, 110];
```

el compilador detectará un problema y mostrará el siguiente mensaje:

**Uso de la variable local no asignada 'adivinatorEdad'**



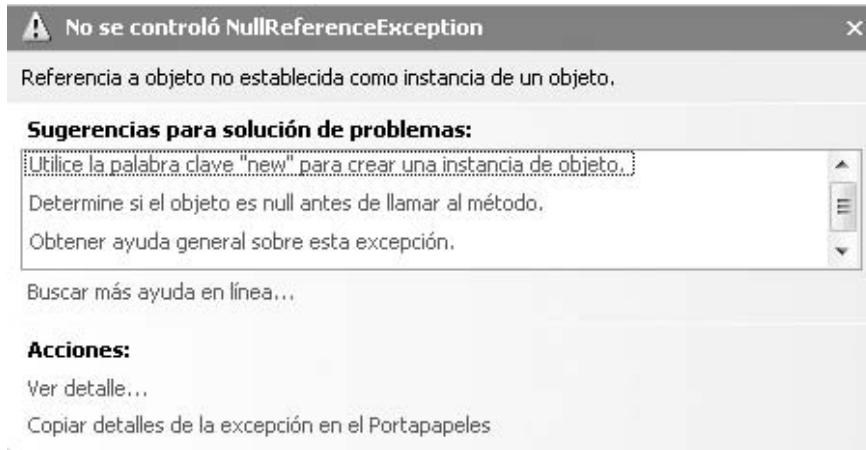


Figura 9.8 Error por intentar el uso de un objeto inexistente.

Hemos tratado de utilizar un objeto que no se ha creado. Lo que falta es una instrucción para crear una instancia de la clase `Random`, como se muestra a continuación:

```
adivinatorEdad = new Random();
```

Ahora bien, si escribimos:

```
Random adivinatorEdad = new Random();
adivinatorEdad = null;
int edad = adivinatorEdad.Next(5, 110);
```

el compilador no podrá detectar que hemos destruido (por error) el objeto al asignarle el valor `null`. Pero al ejecutar el programa obtendremos un mensaje de error como el de la Figura 9.8, y la línea de código que lo contiene quedará resaltada.

Por supuesto, éste es un ejemplo creado a propósito para ilustrar el tema, pero este tipo de errores ocurre con frecuencia en los programas de principiantes.

## Errores lógicos

Los errores lógicos son los más difíciles de encontrar, ya que dependen de la forma en que funciona cada programa, por lo tanto no hay un método automático para detectarlos. Sin embargo, es bueno saber que suelen presentarse dos tipos de errores.

La inicialización implica otorgar un valor inicial a una variable; es fácil cometer el error de no inicializar una variable de manera apropiada. En C# todas las variables de instancia se inicializan de manera automática con cierto valor definido (por ejemplo, los valores `int` se inicializan con cero), aunque tal vez no sea el requerido. Por su parte, las variables locales (las que se declaran dentro de un método) no se inicializan de manera automática; afortunadamente, el compilador marca este error.

También resulta demasiado fácil cometer el error de no indicar el mecanismo para manejar un evento (por ejemplo, el clic de un botón). Esto puede ocurrir si cambiamos el nombre de un componente, como un botón, pero olvidamos modificarlo también en el encabezado del método manejador de eventos.

## Errores comunes de programación

En este capítulo vimos varias causas comunes de errores.

## Nuevas características del IDE

Para insertar un punto de interrupción, haga clic en la barra gris que está a la izquierda de la línea de código correspondiente; para eliminarlo, haga clic en el círculo que lo identifica en esa misma barra. El menú **Depurar** contiene opciones para avanzar paso a paso por los programas.

## Resumen

- Depurar significa encontrar errores (*bugs*) en un programa para poder corregirlos.
- El entorno de desarrollo integrado de C# provee un programa “depurador”.
- Los puntos de interrupción son lugares en donde el programa se detiene temporalmente.
- Avanzar paso a paso significa observar el flujo de ejecución del programa.
- Los valores de las variables pueden mostrarse en puntos de interrupción o durante el avance paso a paso.

## EJERCICIO

- 9.1** Use un programa que haya escrito antes para practicar con el depurador. Coloque puntos de interrupción en el programa, y ejecútelo. Después avance paso a paso por el programa y coloque el cursor sobre los nombres de las variables para ver sus valores.

# 10

## Creación de clases

En este capítulo conoceremos cómo:

- escribir una clase;
- escribir los métodos constructores;
- escribir métodos `public`;
- utilizar variables dentro de un objeto;
- escribir propiedades.

### ● Introducción

---

En capítulos anteriores vimos cómo utilizar las bibliotecas de clases, ya sea seleccionando controles del cuadro de herramientas o codificándolas de manera explícita. En este capítulo veremos de qué manera escribir nuestras propias clases. Una clase describe cualquier cantidad de objetos que pueden fabricarse a partir de ella mediante la palabra clave `new`.

Aprenderemos también que las clases consisten en:

- datos `private` (variables) que contienen información sobre el objeto;
- opcionalmente uno o más métodos constructores, que se utilizan al momento de crear un objeto; su propósito es llevar a cabo cualquier inicialización, por ejemplo, asignando valores iniciales a las variables del objeto;
- métodos `public` que el usuario del objeto puede invocar para llevar a cabo funciones útiles;
- propiedades que nos permiten acceder o modificar las propiedades de un objeto;
- métodos `private`, que se utilizan sólo dentro del objeto y son inaccesibles desde fuera de él.

## ● Cómo diseñar una clase

Cuando el programador piensa en la creación de un nuevo programa, tal vez contemple la necesidad de incluir un objeto que no esté disponible en la biblioteca de clases de C#. Para ejemplificar lo anterior utilizaremos un programa cuyo objetivo es desplegar y manipular un globo, y representaremos el globo como un objeto (un círculo dentro de un cuadro de imagen, como se muestra en la Figura 10.1). Además, pondremos dos botones para cambiar la posición y el tamaño del globo.

Construiremos este programa a partir de dos objetos y, por ende, de dos clases:

- La clase `Globo` proveerá los métodos denominados `Mover` y `CambiarTamaño`, cuyos comportamientos son obvios.
- La clase `Form1` provee la GUI del programa. Esta clase utilizará la clase `Globo` según sea necesario.

Ambas clases se muestran en el diagrama de clases de la Figura 10.2. Estos diagramas representan las clases mediante rectángulos. Cualquier conexión que exista entre ellas se ilustra a través de líneas de unión. En este caso la relación se explica con una anotación arriba de la línea.

Para empezar completaremos la clase `Form1`, y después escribiremos la clase `Globo`.

Como siempre, declaramos las variables de instancia en la parte superior de la clase `Form1` (afuera de los métodos), incluyendo una variable llamada `globo`:

```
private Globo globo;  
private Graphics áreaDibujo;
```

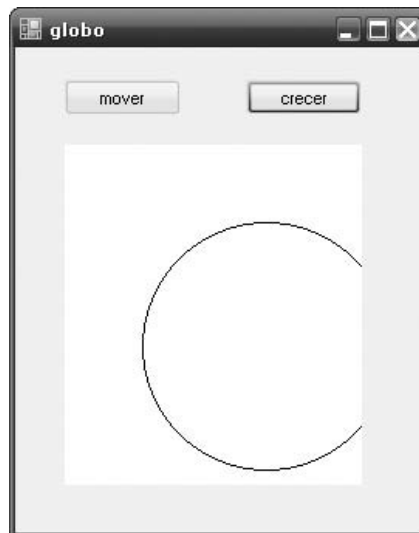


Figura 10.1 El programa Globo.



Figura 10.2 Diagrama de clases que muestra las dos clases del programa Globo.

Dentro del método constructor de `Form1` realizamos la inicialización necesaria, incluyendo la creación de una nueva instancia de la clase `Globo`. Éste es el punto crucial en el que creamos un objeto a partir de nuestra propia clase.

```
globo = new Globo();
áreaDibujo = pictureBox1.CreateGraphics();
```

Ahora viene el código para responder a los eventos de clic de botón:

```
private void botónMover_Click(object sender, EventArgs e)
{
    globo.MoverDerecha(20);
    áreaDibujo.Clear(Color.White);
    globo.Mostrar(áreaDibujo);
}

private void botónCrecer_Click(object sender, EventArgs e)
{
    globo.CambiarTamaño(20);
    áreaDibujo.Clear(Color.White);
    globo.Mostrar(áreaDibujo);
}
```

Éste es todo el código de la clase `Form1`. Escribirlo nos ayuda a entender cómo se utiliza el objeto `globo` y, al mismo tiempo, nos permite ver cuáles métodos y propiedades debe proveer la clase `Globo`, así como la naturaleza de sus parámetros. Esto nos conduce a escribir el código para la clase `Globo`:

```
public class Globo
{
    private int x = 50;
    private int y = 50;
    private int diámetro = 20;

    public void MoverDerecha(int pasoX)
    {
        x = x + pasoX;
    }

    public void CambiarTamaño(int cambio)
    {
        diámetro = diámetro + cambio;
    }

    public void Mostrar(Graphics áreaDibujo)
    {
        Pen lápiz = new Pen(Color.Black);
        áreaDibujo.DrawEllipse(lápiz, x, y, diámetro, diámetro);
    }
}
```

El encabezado de la descripción de la clase empieza con las palabras clave `public` y `class`, y proporciona el nombre de la misma. La descripción completa termina con una llave. La convención de C# (y de casi todos los lenguajes orientados a objetos) es que el nombre de las clases empieza con mayúscula. El cuerpo de la clase consiste en las declaraciones de las variables, los métodos y las propiedades. Observe cómo mejora la legibilidad de la clase gracias al uso de sangría y líneas en blanco. En las siguientes secciones del capítulo veremos con detalle cada uno de los ingredientes que conforman la descripción de la clase `Globo`.

La estructura general de las clases es:

```
public class Globo
{
    // variables de instancia
    // métodos
    // propiedades
}
```

¿En dónde debemos colocar las clases? Una opción consiste en escribir todas las clases y colocarlas en un solo archivo, pero es mejor ponerlas en archivos distintos. El IDE provee una herramienta para ayudarnos a hacerlo. Los pasos son:

1. Seleccione el comando **Agregar clase** del menú **Proyecto**. Enseguida se desplegará la ventana **Agregar nuevo elemento**.
2. Seleccione **Clase** de la lista de plantillas.
3. Sobrescriba el nombre de la clase.
4. Haga clic en el botón **Agregar**.

Este procedimiento crea un archivo distintivo que contendrá el código de la clase. Este archivo es parte del proyecto del programa, de manera que se compila y vincula automáticamente al ejecutarlo.

## ● Variables `private`

---

El globo de nuestro ejercicio anterior tiene asociados ciertos datos: su tamaño (diámetro) y posición (como coordenadas  $x$  y  $y$ ). El objeto globo debe recordar esos valores, lo cual se logra guardándolos en variables que se describen de la siguiente forma:

```
private int x = 50;
private int y = 50;
private int diámetro = 20;
```

Las variables `diámetro`, `x` y `y` se declaran en la parte superior de la clase. Cualquier instrucción dentro de la clase puede acceder a ellas. Se denominan *variables a nivel de clase* o *variables de instancia*.

Como vimos en el capítulo 6, hemos agregado la palabra clave `private` al término que suele utilizarse para introducir las variables (como `int`). Las variables a nivel de clase casi siempre se declaran como `private`. Aunque *podríamos* describir estas variables como `public`, por lo general se considera una mala práctica. Es mejor dejarlas como `private` y usar propiedades o métodos para acceder a sus valores, como veremos más adelante.

## PRÁCTICA DE AUTOEVALUACIÓN

**10.1** Amplíe el objeto globo de manera que tenga una variable que describa el color del globo.

### ● Métodos public

Ciertas características de los objetos deben estar públicamente disponibles para otras piezas del programa. Esto incluye los métodos diseñados para que otros métodos los utilicen. Como hemos visto antes, el globo tiene asociadas ciertas acciones, por ejemplo, la que permite modificar su tamaño. Tales acciones se escriben como métodos. Para cambiar el tamaño utilizamos el siguiente código:

```
public void CambiarTamaño(int cambio)
{
    diámetro = diámetro + cambio;
}
```

Para indicar que está públicamente disponible, debemos anteponer al encabezado del método la palabra clave `public`. El código siguiente constituye el método para mover el globo:

```
public void MoverDerecha(int xPasos)
{
    x = x + xPasos;
}
```

Para completar la clase proveeremos un método que permitirá que el globo se despliegue cada vez que se lo soliciten.

```
public void Mostrar(Graphics áreaDibujo)
{
    Pen lápiz = new Pen(Color.Black);
    áreaDibujo.DrawEllipse(lápiz, x, y, diámetro, diámetro);
}
```

Hemos marcado claramente la diferencia entre los elementos públicamente disponibles y los que son privados. Éste es un importante ingrediente de la filosofía de la programación orientada a objetos. Los datos (variables) y las acciones (métodos) están agrupados, pero de tal forma que parte de la información queda oculta al mundo exterior. Por lo general son los datos los que se ocultan. Esto se denomina *encapsulamiento* u *ocultamiento de información*.

## PRÁCTICAS DE AUTOEVALUACIÓN

**10.2** Escriba un método que mueva el globo hacia arriba por una cantidad que se proporcionará como parámetro. Use el nombre `MoverArriba`.

**10.3** Escriba un método que permita cambiar el color del globo.

**10.4** Reescriba el método `Mostrar`, de manera que despliegue un globo de color.



**Figura 10.3** Estructura de un objeto o clase desde la perspectiva del programador que los escribió.



**Figura 10.4** Estructura de un objeto o clase desde la perspectiva de los usuarios.

Desde la perspectiva del programador encargado de escribirlos, las clases u objetos tienen la estructura general que se muestra en la Figura 10.3, consistente en variables, propiedades y métodos. Sin embargo los usuarios ven el objeto, que les provee un servicio, de manera muy distinta (Figura 10.4): sólo los elementos públicos (por lo general métodos y propiedades) son visibles; todo lo demás está oculto en una caja impenetrable.



## ● Propiedades

Anteriormente comentamos que hacer `public` las variables de instancia constituye una mala práctica de programación. Las propiedades pueden utilizarse como un mecanismo para otorgar a los usuarios un acceso conveniente pero controlado a los datos de un objeto. De hecho, hemos utilizado ya las propiedades de los componentes del cuadro de herramientas y sabemos que, por ejemplo, podemos escribir instrucciones para acceder a los valores de las propiedades de un cuadro de texto:

```
nombre = textBox1.Text;
textBox1.Visible = false;
```

Éstos son ejemplos de cómo acceder a los datos que forman parte de un objeto, pero hay que tener en cuenta que existen dos tipos distintos de acceso:

- leer un valor: a esto se le conoce como acceso *get* (por ejemplo, extraer el texto de un cuadro de texto utilizando la propiedad `Text`);
- escribir el valor: a esto se le conoce como acceso *set* (por ejemplo, cambiar la propiedad `visible` de un cuadro de texto).

A continuación exploraremos cómo proveer este tipo de acceso a los datos. Por ejemplo, suponga que queremos permitir que el usuario de un objeto globo haga referencia (*get*) a la coordenada *x* del globo. Este valor se guarda en la variable llamada `x`, establecida en la parte superior de la clase. Imagine que el usuario del objeto globo (`globo`) quiere ser capaz de extraer y utilizar el valor, como en el siguiente ejemplo:

```
textBox1.Text = Convert.ToString(globo.coordX);
```

Suponga también que, como programadores, deseamos que el usuario pueda cambiar (*set*) el valor de la coordenada *x* con una instrucción como la siguiente:

```
globo.coordX = 56;
```

La manera de proporcionar estas herramientas es mediante una propiedad. He aquí una nueva versión de la clase `Globo`, que incluye el código de la propiedad:

```
public class GloboConPropiedades
{
    private int x = 50;
    private int y = 50;
    private int diámetro = 20;

    public void MoverDerecha(int xPasos)
    {
        x = x + xPasos;
    }

    public void CambiarTamaño(int cambio)
    {
        diámetro = diámetro + cambio;
    }
}
```

```
public void Mostrar(Graphics áreaDibujo)
{
    Pen lápiz = new Pen(Color.Black);
    áreaDibujo.DrawEllipse(lápiz, x, y, diámetro, diámetro);
}

public int coordX
{
    get
    {
        return x;
    }
    set
    {
        x = value;
    }
}
```

El encabezado de las propiedades es similar al que se utiliza en los métodos, excepto que no hay paréntesis para especificar parámetros. La declaración completa termina con una llave de cierre.

La descripción consiste en dos componentes complementarios: uno con `get` y el otro con `set` como encabezados. Cada componente termina con su respectiva llave de cierre. La parte `get` es como un método: devuelve el valor deseado; la parte `set` es como un método: asigna el valor utilizando la palabra clave especial `value`, como se muestra en el ejemplo anterior.

Ahora podemos utilizar estas propiedades en el programa que emplea esta clase. Lo mejoraremos de manera que haya un botón para desplegar el valor de la coordenada `x` en un cuadro de texto. La Figura 10.5 muestra la pantalla resultante.

El siguiente es el código que responde a un clic del botón para mostrar la coordenada `x`; en él se utiliza la propiedad `get` que acabamos de ver:

```
private void mostrarXButton_Click(object sender, EventArgs e)
{
    textBox1.Text = Convert.ToString(globo.coordX);
    globo.Mostrar(áreaDibujo);
}
```

Este programa cuenta también con un botón para modificar la coordenada `x` (que se introduce en el cuadro de texto) mediante la propiedad `set` de la clase `GloboConPropiedades`. El código que responde a un clic en este botón es:

```
private void cambiarXButton_Click(object sender, EventArgs e)
{
    globo.coordX = Convert.ToInt32(textBox1.Text);
    globo.Mostrar(áreaDibujo);
}
```



Figura 10.5 Versión del programa Globo en la que se utilizan propiedades.

Si sólo necesitamos poner a disposición del usuario un mecanismo para ver una propiedad (sin darle la posibilidad de modificar su valor), escribimos la declaración de la propiedad de la siguiente manera:

```
public int coordX
{
    get
    {
        return x;
    }
}
```

Por otra parte, si deseáramos permitirle cambiar un valor pero evitando que pueda verlo, escribiríamos lo siguiente:

```
public int coordX
{
    set
    {
        x = value;
    }
}
```

Tal vez se pregunte por qué es necesario desarrollar un mecanismo tan extenso para usar las propiedades. Sin duda sería más sencillo declarar el valor `x` como `public`, ¿no es así? De esa manera el usuario del objeto sólo tendría que referirse al valor como `globo.x`. Esto es posible, pero constituye una muy mala práctica. En cambio, hay varias razones por las que es preferible utilizar propiedades:

- La clase puede ocultar a los usuarios la representación interna de los datos, sin privarlos de la interfaz externa. Por ejemplo, el autor de la clase `globo` podría optar por mantener las coordenadas del centro de un globo, pero proveer a los usuarios las coordenadas de la esquina superior izquierda del cuadrado que lo contiene.
- El autor de la clase puede optar por restringir el acceso a los datos. Por ejemplo, la clase podría restringir el valor de la coordenada `x` para que tuviera acceso de sólo lectura (`get`), deshabilitando el acceso de escritura (`set`).
- La clase puede validar o comprobar los valores utilizados. Por ejemplo, podría ignorar un intento de proporcionar un valor negativo para una coordenada.

#### PRÁCTICA DE AUTOEVALUACIÓN

**10.5** Escriba una propiedad para permitir que un usuario sólo tenga acceso `get` a la coordenada `y` de un `globo`.

### ● ¿Método o propiedad?

Tanto los métodos como las propiedades proporcionan mecanismos para acceder a un objeto. Entonces ¿cómo elegimos cuál usar? Utilizamos los métodos cuando queremos que un objeto realice cierta acción (por lo general los nombres de los métodos son verbos). En contraste, empleamos propiedades cuando deseamos hacer referencia a cierta información asociada a un objeto (generalmente los nombres de las propiedades son sustantivos).

Esto queda bien ilustrado en las bibliotecas de clases. Por ejemplo, los cuadros de texto tienen los métodos `clear` (borrar) y `appendText` (anexar texto) para realizar acciones, y cuenta con las propiedades `Text` (texto) y `Visible` para referirse al estado del objeto.

Algunos ejemplos de métodos asociados al `globo` de nuestro ejercicio anterior podrían ser: `Mostrar`, `MoverArriba`, `MoverAbajo`, `MoverIzquierda`, `MoverDerecha`, `ReducirTamaño` y `AumentarTamaño`, mientras que algunos ejemplos de propiedades serían: `coordX`, `coordY`, `Diámetro`, `Color` y `Visible`.

En ocasiones se hace necesario elegir entre utilizar una propiedad o crear un método para cumplir un propósito determinado; en este sentido, la decisión es cuestión de estilo. Por ejemplo, para cambiar el color de un componente podríamos crear un método llamado `CambiarColor`, aunque también podríamos usar una propiedad denominada `Color` para el mismo fin.

#### PRÁCTICA DE AUTOEVALUACIÓN

**10.6** ¿Cuáles de los siguientes elementos deberían ser métodos y cuáles propiedades al diseñar una clase llamada `Cuenta` para representar una cuenta bancaria?

`AbonarEnCuenta`, `RetirarDeCuenta`, `SaldoActual`,  
`CalcularInterés`, `Nombre`

## ● Constructores

Al crear un objeto (digamos, globo), es necesario que se de valor a su posición y tamaño. A esto se le conoce como inicializar las variables. Hay dos formas de llevar a cabo esta labor. Una de ellas consiste en incluir la inicialización como parte de la declaración de las variables a nivel de clase:

```
private int x = 50;
private int y = 50;
private int diámetro = 20;
```

Otra manera de inicializar un objeto es escribir un método especial, conocido como *método constructor* o simplemente *constructor* (ya que está involucrado en la construcción del objeto). Este método siempre debe tener el mismo nombre que la clase; no tiene valor de retorno, pero por lo general sí cuenta con parámetros. He aquí un método constructor para la clase `Globo`:

```
public Globo(int xInicial, int yInicial,
             int diámetroInicial)
{
    x = xInicial;
    y = yInicial;
    diámetro = diámetroInicial;
}
```

Este método asigna los valores de los parámetros (el tamaño y la posición) a las variables apropiadas dentro del objeto. Los métodos constructores como éste se escriben en la parte superior de la clase, después de las declaraciones de las variables a nivel de clase.

El método constructor anterior se utilizaría como se muestra en el siguiente ejemplo:

```
Globo globo = new Globo(10, 10, 50);
```

Si el programador no inicializa explícitamente una variable, el sistema de C# le proporcionará un valor predeterminado. Para los números este valor es cero, para las variables `bool` es `false`, para las variables `string` es `""` (una cadena vacía), y para cualquier objeto es `null`. En cualquier caso, se considera que depender de este método de inicialización de variables constituye una mala práctica de programación; es mejor hacer la inicialización de manera explícita, ya sea al declarar la información o mediante instrucciones dentro de un constructor.

Otras acciones que podría realizar un método constructor incluyen la creación de otros objetos que el objeto en cuestión utilice, o la apertura de un archivo que emplee.

Si una clase carece de un constructor explícito, C# asume que tiene un solo constructor con cero parámetros, al cual se le conoce como constructor predeterminado.

## ● Múltiples constructores

Una clase puede tener cero, uno o varios métodos constructores. Si tiene uno o más constructores, por lo general éstos llevan parámetros y deben invocarse con los parámetros apropiados. Por ejemplo, en la clase `Globo` podemos escribir los siguientes dos constructores:

```
public Globo(int xInicial, int yInicial,
             int diámetroInicial)
{
    x = xInicial;
    y = yInicial;
    diámetro = diámetroInicial;
}

public Globo(int xInicial, int yInicial)
{
    x = xInicial;
    y = yInicial;
}
```

Lo cual nos permitiría crear objetos globo de cualquiera de estas formas:

```
Globo globo1 = new Globo(10, 10, 50);
Globo globo2 = new Globo(10, 10);
```

pero no podríamos hacer esto:

```
Globo globo3 = new Globo();
```

Por lo tanto, si escribe varios constructores pero de todas maneras necesita un constructor sin parámetros, tendrá que escribirlo explícitamente, por ejemplo:

```
public Globo()
{
}
```

Ahora tenemos tres constructores para la clase `Globo`; he aquí cómo podríamos usarlos para crear tres objetos distintos a partir de la misma clase:

```
Globo globo1 = new Globo();
Globo globo2 = new Globo(10, 10, 50);
Globo globo3 = new Globo(10, 10);
```

#### PRÁCTICA DE AUTOEVALUACIÓN

---

**10.7** Escriba un método constructor para crear un nuevo globo, especificando únicamente su diámetro.

## ● Métodos `private`

El propósito de escribir una clase radica en permitir la creación de objetos que proporcionen herramientas útiles a otros objetos. Estas herramientas son los métodos y propiedades `public` que ofrece el objeto. Sin embargo, muchas veces las clases incluyen métodos que no es necesario hacer públicos; ése es el caso, de hecho, de los métodos utilizados en la mayoría de los programas de ejemplo que hemos revisado hasta el momento y que, por lo tanto, son `private`.

Ahora bien, suponga que en la clase `Globo` queremos proveer un método que permita a otro objeto averiguar el área de un globo. Entonces podría proporcionarse un método `public` llamado `Área`. No obstante, tal vez decidiéramos que el cálculo del área es un poco complicado y, por ende, que sería mejor delegarlo a un método `private` llamado `CalcÁrea` que sea invocado por `Área` cuando sea necesario. Entonces el método `public` sería el siguiente:

```
public double Área()
{
    double área = CalcÁrea();
    return área;
}
```

Además, creamos un método `private` que actúe como apoyo para los métodos `public` incluidos en la clase:

```
private double CalcÁrea()
{
    double radio;
    radio = diámetro / 2.0;
    return 3.142 * radio * radio;
}
```

Para invocar un método desde el interior del objeto hay que hacer lo siguiente:

```
double área = CalcÁrea();
```

proporcionando el nombre del método y sus parámetros, de la manera usual. Si otro objeto lo invocara habría que poner el nombre de un objeto antes del nombre del método, pero como es invocado por un método que está dentro del mismo objeto, el nombre del objeto que llama al método queda implícito. Si en realidad quisiéramos enfatizar cuál objeto se está usando, podríamos escribir el siguiente código equivalente:

```
double área = this.CalcÁrea();
```

al utilizar la palabra clave `this` estamos haciendo referencia al objeto actual.

Dependiendo de su tamaño y complejidad, una clase podría tener varios métodos `private`. Su propósito es clarificar y simplificar el funcionamiento de la clase.

## ● Operaciones sobre objetos

Muchos de los objetos que se utilizan en programas de C# deben declararse como clases, pero no todos. Al declarar variables utilizamos `int`, `bool`, `string` y `double`, que se conocen como tipos *predefinidos*. Estos tipos predefinidos están incluidos como parte del lenguaje C#. Mientras que los

nombres de las clases suelen empezar con mayúscula, los nombres de estos tipos predefinidos comienzan con minúscula. Cualquier variable de tipo predefinido puede utilizarse tan pronto como sea declarada. Por ejemplo:

```
int número;
```

Esta instrucción declara la variable `número`, y la crea al instante. En contraste, la creación de cualquier otro objeto tiene que realizarse de manera explícita mediante el uso de `new` (o gráficamente, arrastrando la clase del cuadro de herramientas). Por ejemplo:

```
Globo globo = new Globo(10, 20, 50);
```

Así, en C# las variables pueden ser:

- tipos predefinidos como `int`, `bool` y `double`, u
- objetos creados explícitamente a partir de clases, ya sea arrastrándolas del cuadro de herramientas, o mediante el uso de la palabra clave `new`.

Las variables declaradas de tipo predefinido incluyen de antemano una colección completa de acciones que podemos realizar con ellas. Por ejemplo, con las variables de tipo `int` tenemos la posibilidad de:

- declarar variables;
- asignar valores utilizando `=`;
- realizar operaciones aritméticas;
- comparar mediante el uso de `==`, `<`, etc.;
- usarlas como parámetro o valor de retorno.

El programador no puede hacer necesariamente todas esas tareas con objetos. Muchos de los elementos empleados en los programas de C# son objetos pero, como hemos visto, no todos lo son. Es tentador suponer que podemos usar todas estas operaciones con cualquier objeto, pero no es así. ¿Qué podemos hacer con un objeto? Al escribir una clase definimos el conjunto de operaciones que pueden realizarse sobre los objetos de ese tipo. Por ejemplo, con la clase `Globo` hemos definido las operaciones `CambiarTamaño`, `Mover` y `Mostrar`. El programador no debe suponer que puede hacer cualquier otra cosa con los objetos de tipo `globo`. Sin embargo, para cualquier objeto se puede confiar en su capacidad de hacer lo siguiente:

- crearlos;
- usarlos como parámetros y como valores de retorno;
- asignarlos a una variable de la misma clase mediante el uso de `=`;
- usar los métodos y propiedades que se proporcionan como parte de su clase.

## PRÁCTICA DE AUTOEVALUACIÓN

**10.8** Escriba una lista de operaciones que puedan realizarse con un objeto de la clase `Globo`, y dé ejemplos de cómo usarlas.



## ● Destrucción de objetos

Ya vimos cómo crear objetos mediante el uso de la poderosa palabra `new`. ¿Pero cómo deshacernos de ellos? Una respuesta obvia y certera sería que dejan de funcionar cuando el programa termina de ejecutarse. También es cierto que lo hacen cuando el programador deja de utilizarlos. Por ejemplo, si hacemos lo siguiente para crear un nuevo objeto:

```
Globo globo;
globo = new Globo(20, 100, 100);
```

y después:

```
globo = new Globo(40, 200, 200);
```

lo que ocurre es que el primer objeto creado con `new` tiene una existencia muy corta, ya que desaparece cuando el programa ya no tiene conocimiento de él y su valor es usurpado por el objeto más reciente. Cuando se destruye un objeto, la memoria que se utilizaba para almacenar los valores de sus variables y cualquier otro recurso es reclamada para que el sistema en tiempo de ejecución la emplee en otros procesos. A esto se le conoce como *recolección de basura*. En C# la recolección de basura es automática (a diferencia de lo que ocurre en otros lenguajes, como C++, en los que el programador tiene que llevar cuenta de los objetos que ya no son necesarios).

Por último, podemos destruir un objeto al asignarle el valor `null`; por ejemplo:

```
globo = null;
```

La palabra clave `null` de C# describe un objeto no existente (no instanciado).

## ● Métodos y propiedades `static`

Algunos métodos no necesitan un objeto para trabajar. Por ejemplo, métodos matemáticos como los utilizados para obtener la raíz cuadrada (`Sqrt`) y el seno de un ángulo (`Sin`) se proporcionan dentro de una biblioteca de clases llamada `Math`. Para utilizarlos en un programa escribimos instrucciones como:

```
double x, y;
x = Math.Sqrt(y);
```

En esta instrucción hay dos variables `double` llamadas `x` y `y`, pero no hay objetos. Tenga en cuenta que `Math` es el nombre de una clase, y no de un objeto. El método de raíz cuadrada `Sqrt` actúa sobre su parámetro `y`. La pregunta es: si `Sqrt` no es método de un objeto, entonces ¿qué es? La respuesta es que los métodos como éste forman parte de una clase, pero se describen como `static`. Al utilizar uno de estos métodos hay que anteponer a su nombre el de la clase a la que pertenece.

La clase `Math` tiene la siguiente estructura, en la que los métodos se etiquetan como `static`:

```

public class Math
{
    public static double Sqrt(double x)
    {
        // cuerpo de Sqrt
    }

    public static double Sin(double x)
    {
        // cuerpo de Sin
    }
}

```

`ToInt32` es otro ejemplo de un método `static`, en este caso, dentro de la clase `Convert`. Un ejemplo de una propiedad `static` se encuentra en la clase `Color`: los diversos colores (como `Color.White`, `Color.Black`, etc.) están disponibles para que otras clases los utilicen.

¿Cuál es el propósito de los métodos `static`? En la programación orientada a objetos todo se escribe como parte de una clase; no existe nada fuera de ellas. Pensemos en la clase `Globalo`; ésta contiene variables `private`, como `x` y `y`, que registran el estado de un objeto. En contraste, los métodos independientes como `Sqrt` no involucran un estado y no forman obviamente parte de una clase; no obstante, deben obedecer la regla central de la programación orientada a objetos: formar parte de una clase. Tal es la razón de ser de los métodos `static`. Así, es común que los programadores utilicen las propiedades y métodos `static` de biblioteca, pero es muy poco usual que los programadores novatos los escriban.

#### PRÁCTICA DE AUTOEVALUACIÓN

**10.9** El método `static Max`, que se encuentra en la clase `Math`, encuentra el máximo de sus dos parámetros `int`. Escriba un ejemplo de invocación a `Max`.

## Fundamentos de programación

La programación orientada a objetos se basa, precisamente, en la construcción de programas a partir de objetos. Los *objetos* son combinaciones de ciertos datos (variables) y ciertas acciones (métodos) que desempeñan funciones útiles en un programa. El programador diseña un objeto de manera que los datos y las acciones estén estrechamente relacionados entre sí en vez de agruparlos al azar.

Al igual que en casi todos los lenguajes basados en la programación orientada a objetos, en C# no es posible escribir de manera directa instrucciones que describan un objeto. En vez de ello, el lenguaje hace que el programador defina todos los objetos de la misma clase. Por ejemplo, si necesitamos un objeto botón, vamos al cuadro de herramientas y seleccionamos la clase `Button`, arrastramos una instancia de esa clase y la colocamos en el formulario. Si necesitamos un segundo botón, creamos una segunda instancia de esa misma clase. La descripción de la estructura de todos los posibles botones se llama *clase*, y constituye una plantilla o plano maestro para fabricar cualquier número de objetos. En consecuencia, podemos decir que una clase es la generalización de un objeto.

El concepto de clases es común en casi todas las actividades de diseño. Por lo general, antes de construir un objeto real hay que crear un diseño del mismo. Esto se aplica en el diseño automotriz, en la arquitectura, en la construcción, e incluso en las bellas artes. Se bosqueja cierto tipo de plano en papel o en un programa de computación, y el diseño resultante especifica todos los detalles del objeto deseado, de manera que si el diseñador es arrollado por un autobús alguien más pueda llevar a cabo su construcción. Una vez que se ha diseñado un objeto (digamos un automóvil, un libro o una computadora), es posible construir varias instancias idénticas. Por lo tanto, el diseño especifica la composición de cualquier cantidad de objetos. Lo mismo ocurre en el ámbito de la programación orientada a objetos: una clase es el plano para construir cualquier número de objetos idénticos. Una vez que especificamos una clase, podemos hacer cualquier cantidad de objetos con el mismo comportamiento.

Analicemos de nuevo la clase `Button`: lo que tenemos es la descripción de la apariencia de cada objeto botón. Los botones sólo difieren en sus propiedades individuales, como sus posiciones en el formulario. Por ello, en la programación orientada a objetos una clase es la especificación para cualquier cantidad de objetos iguales. Una vez que se describe una clase puede construirse un objeto específico al crear una *instancia* de esa clase. Es algo así como decir que hemos tenido una instancia de gripe en el hogar, o que cierto Ford modelo T es una instancia del diseño original de ese automóvil. Su propia cuenta bancaria es una instancia de la clase cuenta bancaria.

Los objetos son agrupamientos lógicos de variables, métodos y propiedades. Cada uno constituye un módulo autocontenido que se puede utilizar y entender con facilidad. El principio del ocultamiento o encapsulamiento de información implica que los usuarios de un objeto tienen una vista restringida del mismo. El objeto proporciona un conjunto de servicios en forma de métodos y propiedades `public` que otros pueden utilizar. El resto del objeto, sus variables y las instrucciones que implementan los métodos, están ocultos. Esto mejora la abstracción y la modularidad.

En programación el término *accesibilidad* (algunas veces conocido como *reglas de alcance* o *visibilidad*) implica la normatividad para acceder a las variables y los métodos. Desde nuestra perspectiva, las reglas de accesibilidad son equivalentes a la norma que establece que en Australia debemos conducir por la izquierda, o a la que determina que sólo podemos entrar a una casa ajena por la puerta delantera. En los programas el compilador se encarga de hacer cumplir al pie de la letra las reglas de este tipo, para evitar un acceso deliberado o erróneo a la información protegida. Las reglas de accesibilidad restringen al programador, pero al mismo tiempo lo ayudan a organizar sus programas de manera clara y lógica. Las reglas de accesibilidad asociadas a las clases y los métodos permiten que el programador encapsule convenientemente tales elementos.

El programador tiene la libertad de describir cada variable, método y propiedad como `public` o `private`. Cualquier instrucción en cualquier parte de una clase puede invocar a cualquier método, ya sea `public` o `private`. Además, cualquier instrucción puede hacer referencia a cualquier variable de instancia. La excepción es que las variables locales (las que se declaran dentro de un método) sólo pueden ser utilizadas dentro del método en el que se declararon.

Cuando una clase hace referencia a otra, únicamente aquellos métodos, propiedades y variables etiquetados como `public` pueden ser utilizados fuera de ella; todos los demás elementos son inaccesibles. Es una buena práctica de diseño minimizar el número de métodos y propiedades `public`, restringiéndolos de manera que sólo se ofrezcan los servicios de la clase. También es recomendable nunca (o muy pocas veces) declarar las variables como `public`. Si debemos inspeccionar o modificar una variable, es mejor que haya un método o propiedad para realizar ese trabajo.

En resumen, una variable, método o propiedad dentro de una clase puede describirse en estos términos:

1. Si es **public** puede emplearse en cualquier parte (desde el interior de la clase o desde cualquier otra clase).
2. Si es **private** puede utilizarse sólo dentro de la clase en la que se declaró.

En computación algunas veces a las clases se les denomina *tipo de datos abstracto* (ADT). Un tipo de datos es una clase de variable, por ejemplo **int**, **double** o **bool**. Estos tipos predefinidos están integrados en el lenguaje C#, disponibles para utilizarlos de inmediato. Además, cada uno de esos tipos cuenta con un conjunto de operaciones asociado. Por ejemplo, con un **int** podemos realizar operaciones de asignación, suma, resta, etcétera. La clase **Globo** que vimos antes es un ejemplo de ADT: define ciertos datos (variables) junto con una colección de operaciones (métodos) y propiedades capaces de realizar operaciones con los datos. La clase presenta una abstracción de un globo; los detalles concretos de la implementación están ocultos.

Ahora podemos entender de manera integral la estructura general de un programa. Analice el código que se crea de manera automática cuando abrimos una nueva aplicación de Windows en el entorno de desarrollo integrado de C#:

```
public partial class Form1
```

Como puede ver, ésta es la descripción de una clase llamada **Form1** ya que, como todo lo demás, los formularios son una clase. La palabra **partial** que aparece en la descripción de la clase indica que su definición se divide en varios archivos de código fuente: **Form1.cs**, **Program.cs** y **Form1.Designer.cs**. El IDE de C# se encarga de esto por nosotros. Cuando un programa empieza su ejecución, invoca el método **Main**, que se encuentra en el archivo **Program.cs**. Como vemos, se ejecuta la siguiente instrucción:

```
new Form1()
```

Esta operación crea un objeto a partir de esta clase. El objeto en sí crea los demás objetos que sean necesarios, como los botones.

## Errores comunes de programación

- En ocasiones los programadores principiantes quieren codificar un objeto directamente. Esto no es posible; para lograrlo debemos declarar una clase y después crear una instancia de esa clase.
- No olvide inicializar las variables de instancia de manera explícita, por medio de un método constructor o como parte de la declaración misma; no deje que C# inicialice las variables con valores predeterminados.

Si usted declara:

```
Globo globoRojo;
```

y después ejecuta la línea de código:

```
globoRojo.Mostrar(áreaDibujo);
```

su programa terminará con un mensaje de error indicando que hay una excepción de apuntador a **null**. Esto se debe a que declaró un objeto, pero no lo creó (con **new**). El objeto **globoRojo** no

existe; dicho de manera más precisa, tiene el valor `null`. En la programación elemental casi nunca utilizamos el valor `null`, a menos que olvidemos utilizar `new`.

## Secretos de codificación

- Las clases tienen la siguiente estructura:

```
public class NombreClase
{
    // declaraciones de las variables de instancia
    // declaraciones de los métodos y las propiedades
}
```

- Las variables, los métodos y las propiedades pueden describirse como `public` o `private`.
- Uno o más de los métodos que forman parte de una clase pueden tener el mismo nombre que ésta. Es posible invocar cualquiera de estos métodos constructores (con los parámetros apropiados) para inicializar el objeto al momento de crearlo.
- La declaración de un método público tiene la estructura siguiente:

```
public void NombreMétodo(parámetros)
{
    // cuerpo
}
```

- La declaración de una propiedad tiene esta estructura:

```
public int NombrePropiedad
{
    get
    {
        // instrucciones para devolver un valor
    }
    set
    {
        // instrucciones para asignar un valor a una variable
    }
}
```

- Se puede declarar una propiedad sin la parte `set` o sin la parte `get`.
- Las propiedades o métodos independientes llevan el prefijo `static` en su encabezado.
- Para invocar un método estático de una clase debe emplearse una línea como ésta:

```
NombreClase.NombreMétodo(parámetros);
```

## Nuevos elementos del lenguaje

- **class** Aparece en el encabezado de la descripción de una clase.
- **public** La descripción de una variable, método o propiedad que se puede utilizar desde cualquier parte.
- **private** La descripción de una variable, método o propiedad que sólo puede utilizarse dentro de la clase.
- **new** Se utiliza para crear una nueva instancia de una clase (un nuevo objeto).
- **set** Introduce la parte de la declaración de una propiedad que modifica el valor de ésta.
- **get** Introduce la parte de la declaración de una propiedad que devuelve un valor.
- **value** El nombre del valor que se va a asignar en una propiedad **set**.
- **this** El nombre del objeto actual.
- **null** El nombre de un objeto que no existe.
- **static** La descripción que se adjunta a una variable, propiedad o método que pertenecen a una clase como un todo y no a una instancia creada como un objeto de la clase.

## Resumen

- Los objetos son colecciones de datos, y las acciones, métodos y propiedades asociadas que pueden actuar sobre esos datos. Los programas de C# se construyen como un grupo de objetos.
- Los métodos que tienen el mismo nombre que la clase llevan a cabo la inicialización de un objeto recién creado. Estos métodos se denominan constructores.
- Los elementos que conforman una clase pueden declararse como **private** o **public**. Un elemento **private** sólo se puede utilizar dentro de la clase; un elemento **public** puede emplearse en cualquier parte (dentro o fuera de la clase). Al diseñar un programa de C#, por lo general se utiliza la menor cantidad posible de variables **public** para mejorar el ocultamiento de información.
- La descripción **static** significa que la variable, propiedad o método pertenece a la clase y no a un objeto específico. Los métodos **static** se pueden invocar de manera directa, sin necesidad de crear una instancia de la clase mediante **new**. Los métodos o propiedades **static** son útiles cuando no necesitan estar asociados a un objeto específico, o para llevar a cabo acciones para la clase en general.

## EJERCICIOS

- 10.1 Globos** Agregue varios datos más a la clase **Globo**: una variable **string** que guarde el nombre del globo, y una variable que describa su color. Añada código para inicializar estas variables mediante el uso de un método constructor, y las instrucciones necesarias para desplegarlos.

Mejore el programa Globo con botones que muevan el globo a la izquierda, a la derecha, hacia arriba y hacia abajo.

- 10.2 Pantalla de amplificador** Algunos amplificadores estereofónicos cuentan con un dispositivo visual que muestra el volumen de salida. Ese dispositivo aumenta y disminuye su nivel de acuerdo con el volumen en cualquier momento dado. De igual manera, ciertas pantallas tienen indicadores que muestran los valores máximo y mínimo alcanzados desde que el amplificador se encendió.

Escriba un programa que muestre en cuadros de texto los valores máximo y mínimo a los que se haya establecido una barra de seguimiento.

Escriba la pieza de código que recuerde los valores y los compare en una clase. Esta clase debe tener un método llamado `NuevoValor` junto con las propiedades `MenorValor` y `MayorValor`.

- 10.3 Cuenta bancaria** Escriba un programa que simule una cuenta bancaria. Un cuadro de texto debe permitirnos realizar depósitos (un número positivo) en la cuenta y hacer retiros (un número negativo) de la misma. El estado de la cuenta debe mostrarse de manera continua, y si entra en números rojos (saldo negativo) desplegar un mensaje apropiado. Cree una clase llamada `Cuenta` para representar cuentas bancarias. Debe tener los métodos `Depositar` y `Retirar`, junto con una propiedad llamada `SaldoActual`.

- 10.4 Guardar puntuación** Diseñe y escriba una clase que sirva para guardar la puntuación para un juego de computadora. Debe mantener un solo entero: la puntuación. Además es necesario que proporcione métodos para inicializar la puntuación en cero, para incrementar la puntuación, para reducir la puntuación, y para devolver la puntuación. Escriba instrucciones para crear y utilizar un solo objeto.

- 10.5 Dados** Diseñe y escriba una clase que actúe como un dado que se pueda lanzar para obtener un valor del 1 al 6. Primero escriba la clase de manera que siempre obtenga el valor 6. Cree un programa para crear y utilizar un objeto dado. La pantalla debe mostrar un botón que al ser oprimido “lance” el dado y despliegue su valor.

Después modifique la clase dado, de manera que proporcione un valor un punto mayor al que tenía la última vez que se lanzó; por ejemplo, 4 si la última ocasión cayó en 3.

Luego transforme una vez más la clase, de manera que utilice el generador de números aleatorios de la biblioteca.

Algunos juegos, como el backgammon y el monopolio, requieren dos dados. Escriba instrucciones de C# para crear dos instancias del objeto dado, lanzarlos y mostrar los resultados.

- 10.6 Generador de números aleatorios** Escriba su propio generador de números aleatorios, creando para ello una clase que utilice una fórmula para obtener el siguiente número pseudoaleatorio a partir del anterior. Los programas de números aleatorios funcionan empezando con cierto valor de “semilla”. A partir de ese momento, el número aleatorio actual se utiliza como base para obtener el siguiente. Esto se lleva a cabo realizando ciertos cálculos con el número actual para convertirlo en alguno otro (aparentemente aleatorio). Una fórmula que podemos usar para los enteros es:

```
siguienteA = ((anteriorA * 25173) + 13849) % 65536;
```

Esta fórmula produce números en el rango de 0 a 65,535. Los números específicos derivados de la misma han demostrado su capacidad para producir buenos resultados de tipo aleatorio.

**10.7 Números complejos** Escriba una clase llamada `Complejo` para representar números complejos (junto con sus operaciones). Los números complejos consisten de dos partes: una parte real (un `double`) y una imaginaria (un `double`). El método constructor debe crear un nuevo número complejo mediante el uso de los valores `double` que se proporcionen como parámetros, de la siguiente forma:

```
Complejo c = new Complejo(1.0, 2.0);
```

Escriba las propiedades `Real` e `Imaginaria` para obtener las partes respectivas de un número complejo, mismas que deben utilizarse de la siguiente manera:

```
double x = c.Real;
```

Cree un método para sumar dos números complejos y devolver el resultado. La parte real es la suma de las dos partes reales; la parte imaginaria es la suma de las dos partes imaginarias. Una invocación al método tendría esta forma:

```
Complex c = c1.Sum(c2);
```

Escriba un método para calcular el producto de dos números complejos. Si un número tiene los componentes  $x_1$  y  $y_1$ , y el segundo tiene los componentes  $x_2$  y  $y_2$ :

- la parte real del producto =  $x_1 \times x_2 - y_1 \times y_2$
- la parte imaginaria del producto =  $x_1 \times y_2 + x_2 \times y_1$

#### SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

**10.1**    `private Color color;`

**10.2**    `public void MoverArriba(int cantidad)`  
           {  
               `coordY = coordY - cantidad;`  
           }

**10.3**    `public void CambiarColor(Color nuevoColor)`  
           {  
               `color = nuevoColor;`  
           }

**10.4**    `public void Mostrar(Graphics áreaDibujo)`  
           {  
               `Pen lápiz = new Pen(color);`  
               `áreaDibujo.DrawEllipse(lápiz, x, y, diámetro, diámetro);`  
           }



```
10.5    public int CoordY
        {
            get
            {
                return y;
            }
        }
    }
```

**10.6** Métodos: AbonarCuenta, RetirarCuenta, CalcularInterés  
Propiedades: SaldoActual, Nombre

```
10.7    public Globo(int diámetroInicial)
        {
            diámetro = diámetroInicial;
        }
    }
```

**10.8** Los métodos son: CambiarColor, MoverArriba, MoverDerecha, CambiarTamaño, Mostrar  
Ejemplos:

```
globo.CambiarColor(Color.Red);
globo.MoverArriba(20);
globo.MoverDerecha(50);
globo.CambiarTamaño(10);
globo.Mostrar(áreaDibujo);
```

Las propiedades son: CoordX, CoordY, Diámetro  
Ejemplos:

```
globo.CoordX = 20;
int y = globo.CoordY;
globo.Diámetro = 50;
```

```
10.9    int x;
        x = Math.Max(7, 8);
```



# Herencia

**En este capítulo conoceremos cómo:**

- crear una nueva clase a partir de una existente, mediante el uso de la herencia;
- declarar variables `protected` (protegidas), y cuándo hacerlo;
- utilizar la redefinición y en qué momento hacerlo;
- dibujar un diagrama de clases que describa la herencia;
- utilizar la palabra clave `base`;
- utilizar clases abstractas y `abstract`.

## ● Introducción

---

Los programas se construyen a partir de objetos, los cuales son instancias de clases. Algunas clases forman parte de la biblioteca de C#, y otras son escritas por el programador. Cuando empezamos a escribir un nuevo programa buscamos clases útiles en la biblioteca, y revisamos aquellas que hemos escrito en el pasado. Esta metodología de programación orientada a objetos implica que en vez de empezar los programas desde cero, para su creación nos basamos en el trabajo anterior. No obstante, es común encontrar una clase que parezca útil y que haga casi todo lo que necesitamos, pero que no sea exactamente lo que queremos. La herencia es una manera de resolver este problema: gracias a esta característica podemos usar una clase existente como base para crear una clase modificada.

Considere la siguiente analogía. Suponga que desea comprar un nuevo automóvil y para ello se dirige a una agencia automotriz en donde hay una variedad de modelos producidos en serie. A usted le gusta uno en especial, pero eso no significa que tenga la característica especial que está buscando. Al igual que las clases, el automóvil en cuestión fue fabricado a partir de planos que describen muchos automóviles idénticos. Si el mundo automotriz contara con una función similar a la herencia (disponible en el ámbito de la programación), podría solicitar la creación de un automóvil que tuviera todas las particularidades de los producidos en serie, pero con los cambios adicionales que usted requiere.

## ● Uso de la herencia

Comenzaremos el tema analizando una clase similar a la que ya hemos usado varias veces a lo largo de este libro. Se trata de la clase para representar una esfera. Las esferas tienen un radio y una posición en el espacio. Al desplegarse en pantalla, la esfera debe aparecer como un círculo (el método para mostrar una esfera simplemente invoca el método de biblioteca `DrawEllipse`). El diámetro de la esfera está fijo en 20 píxeles. Sólo hemos modelado las coordenadas  $x$  y  $y$  de una esfera (y no la coordenada  $z$ ), debido a que vamos a mostrar una representación bidimensional en la pantalla.

He aquí la descripción de la clase para una esfera. Por lo general esta clase se coloca en su propio archivo, como vimos en el capítulo 10.

```
public class Esfera
{
    protected int coordX = 100, coordY = 100;
    protected Pen lápiz = new Pen(Color.Black);

    public int X
    {
        set
        {
            coordX = value;
        }
    }

    public int Y
    {
        set
        {
            coordY = value;
        }
    }

    public virtual void Mostrar(Graphics áreaDibujo)
    {
        áreaDibujo.DrawEllipse(lápiz, coordX, coordY, 20, 20);
    }
}
```

Cabe mencionar que en este programa hay varios elementos nuevos, incluyendo las palabras clave `protected` y `virtual`. Esto se debe a que escribimos la clase de manera que pueda utilizarse para poner en acción la característica de la herencia. En el transcurso de este capítulo veremos lo que significan estos nuevos elementos.

Supongamos que alguien escribió y evaluó esta clase, y la puso a disposición de otros usuarios. Ahora tenemos que escribir un nuevo programa y necesitamos una clase muy parecida a ésta, sólo que la emplearemos para describir burbujas. Esta nueva clase, llamada `Burbuja`, nos permitirá realizar acciones adicionales: modificar el tamaño de una burbuja y moverla verticalmente. La li-

mitación de la clase `Esfera` es que describe objetos que no se mueven, y cuyo tamaño no se puede modificar. En consecuencia, requerimos una propiedad adicional que nos permita establecer un nuevo valor para el radio de la burbuja. Podemos hacer esto sin alterar la clase existente; para ello hay que escribir una clase diferente que utilice el código que ya se encuentra en la clase `Esfera`. En este sentido, decimos que la nueva clase “hereda” las variables, las propiedades y los métodos de la otra, o que la nueva clase es una subclase de la anterior. A la clase anterior se le llama superclase de la nueva clase. He aquí cómo escribir la nueva clase:

```
public class Burbuja : Esfera
{
    protected int radio = 10;

    public int Tamaño
    {
        set
        {
            radio = value;
        }
    }

    public override void Mostrar(Graphics áreaDibujo)
    {
        áreaDibujo.DrawEllipse(lápiz, coordX, coordY,
                               2 * radio, 2 * radio);
    }
}
```

Esta nueva clase tiene el nombre `Burbuja`. El signo de dos puntos y la mención de la clase `Esfera` indican que `Burbuja` es heredera de la clase `Esfera` o, en otras palabras, que `Burbuja` es una subclase de `Esfera`. Esto significa que `Burbuja` hereda todos los elementos que no estén descritos como `private` dentro de la clase `Esfera`. En las siguientes secciones exploraremos las demás características de esta clase.

### ● `protected`

---

Cuando usamos la herencia, `private` es un término demasiado privado y `public` resulta demasiado público. Si una clase necesita dar a sus subclases acceso a ciertas variables, propiedades o métodos específicos, pero debe evitar que otras clases accedan a ellos, puede etiquetarlos como `protected`. Si hiciéramos una analogía con una familia, diríamos que una madre permite que sus hijos utilicen su automóvil, pero sólo les otorga ese permiso a ellos.

Volviendo a la clase `Esfera`, necesitamos variables para describir las coordenadas que ocupa:

```
private int coordX, coordY;
```

Ésta es una buena manera de hacerlo, pero debe haber una mejor. Podría darse el caso de que en el futuro alguien escribiera una clase que heredara los atributos de ésta y proporcionara un método adicional para mover una esfera. Este método necesitaría acceso a las variables `coordX` y `coordY` que, por desgracia, son inaccesibles ya que se etiquetaron como `private`. Por lo tanto, para anticiparnos a este posible uso en el futuro, sería mejor tomar la decisión de etiquetarlas como `protected`:

```
protected int coordX, coordY;
```

Esta declaración protege las variables de manera que otras clases no puedan hacer mal uso de ellas, pero permite que accedan a ellas ciertas clases privilegiadas: las subclasses.

Suponga que declaramos estas variables `private`, como lo teníamos planeado en un principio. La consecuencia es que de esa manera sería imposible reutilizar la clase. La única opción sería editarla y reemplazar la descripción `private` por `protected` para esos elementos específicos, pero eso viola uno de los principios de la programación orientada a objetos: alterar una clase existente que haya sido probada y utilizada está prohibido. En consecuencia, al escribir una clase debemos esforzarnos por tener en cuenta los posibles usos que se le darán en el futuro. El programador que escribe una clase siempre lo hace con la esperanza de que alguien la reutilice y la amplíe: éste es otro de los principios de la programación orientada a objetos. El uso cuidadoso de `protected` en vez de `public` o `private` nos puede ayudar a hacer que una clase sea más atractiva para poner en práctica las ventajas de la herencia.

En resumen, los cuatro niveles de accesibilidad de una variable, propiedad o método en una clase son:

1. **public**: puede utilizarse en cualquier parte. Como regla, cualquier propiedad o método que ofrezca un servicio a los usuarios de una clase se debe etiquetar como `public`.
2. **protected**: puede emplearse dentro de esta clase y desde cualquier subclase.
3. **private**: sólo es posible utilizar este nivel dentro de la clase que se está declarando. Por lo general las variables de instancia deben declararse como `private` y algunas veces como `protected`.
4. Las variables locales (que se declaran dentro de un método específico) nunca podrán utilizarse fuera del mismo.

Así pues, una clase puede tener un acceso controlado a su superclase inmediata y a las superclases que estén arriba de ella en la jerarquía de clases, como si éstas formaran parte de la clase en sí. Si recurrimos a la analogía de la familia, es como poder gastar libremente el dinero de nuestra madre o de cualquiera de sus antepasados, siempre y cuando lo hayan puesto en una cuenta etiquetada como `public` o `protected`. Quienes no forman parte de la familia sólo pueden acceder al dinero `public`.

## ● Elementos adicionales

Una interesante forma de construir una nueva clase a partir de otra es mediante la inclusión de variables, propiedades y métodos adicionales.

Como se muestra en el ejemplo siguiente, la nueva clase `Burbuja` declara una variable y una propiedad adicionales:

```
protected int radio = 10;

public int Tamaño
{
    set
    {
        radio = value;
    }
}
```

La nueva variable es **radio**, y se añade a las variables existentes en **Esfera** (**coordX** y **coordY**). Por lo tanto, el número de variables se ha ampliado. La nueva clase también tiene la propiedad **Tamaño**, además de las propiedades que ya incluía **Esfera**.

#### PRÁCTICA DE AUTOEVALUACIÓN

**11.1** Un objeto pelota es como un objeto **Esfera**, pero tiene características adicionales: es capaz de moverse a la izquierda y a la derecha. Escriba una clase llamada **Pelota** que herede la clase **Esfera** pero proporcione los métodos adicionales **MoverIzquierda** y **MoverDerecha**.

### ● Redefinición

Otra característica de la nueva clase **Burbuja** es una versión modificada del método **Mostrar**:

```
public override void Mostrar(Graphics áreaDibujo)
{
    áreaDibujo.DrawEllipse(lápiz, coordX, coordY,
                           2 * radio, 2 * radio);
}
```

Esto es necesario debido a que la nueva clase tiene un radio que se puede modificar, mientras que en la clase **Esfera** el radio estaba fijo. Esta nueva versión de **Mostrar** en **Burbuja** sustituye la versión que teníamos en la clase **Esfera**. En términos de programación, decimos que la nueva versión redefine o sobreescribe la versión anterior. Esta última tiene la descripción virtual, y la nueva la descripción **override**.

No confunda la redefinición (*override*) con la sobrecarga (*overload*) que comentamos en el capítulo 5 al hablar sobre los métodos:

- Sobrecargar significa escribir un método (en la misma clase) que tiene el mismo nombre que otro, pero distintos parámetros.
- Redefinir significa escribir un método en una subclase, de manera que tenga el mismo nombre y los mismos parámetros que otro.

En resumen, en la clase que hereda:

- creamos una variable adicional;
- creamos una propiedad adicional;
- redefinimos un método (proporcionamos un método que se utilizará en vez del método que ya existía).

Veamos ahora qué hemos logrado hasta el momento. Teníamos una clase llamada **Esfera**. Requeríamos una nueva clase llamada **Burbuja** que fuera similar a **Esfera**, pero necesitábamos características adicionales. En consecuencia, para crear la nueva clase ampliamos las características de la que ya teníamos. Hemos explotado al máximo lo que tenían en común las dos clases, y evitamos volver a escribir piezas del programa que ya existían. Por supuesto, las dos clases que escribimos (**Esfera** y **Burbuja**) están a nuestra disposición.

Siguiendo con las analogías familiares la herencia, tal como la hemos planteado en el ejemplo anterior, significa que podemos gastar nuestro propio dinero, y también el de nuestra madre.

Cabe mencionar que es posible redefinir variables, esto es, declarar variables en una subclase que redefinen variables de la superclase. No hablaremos más sobre el tema por dos razones: la primera es que nunca tendremos la necesidad de hacer esto, y la segunda es que constituye una muy mala práctica de programación. Al crear una subclase a partir de una clase (es decir, al hacerla heredera de ésta), sólo tendremos que:

- agregar métodos y/o propiedades adicionales;
- agregar variables adicionales;
- redefinir métodos y/o propiedades.

## ● Diagramas de clases

Una buena forma de visualizar la herencia es mediante el uso de un diagrama de clases como el que se muestra en la Figura 11.1. Ahí podemos ver que **Burbuja** es una subclase de **Esfera**, la cual a su vez es una subclase de **Object**. Cada clase se muestra como un rectángulo. Las flechas que hay entre las clases denota una relación de herencia, apuntando de la subclase a la superclase.

Las clases en la biblioteca o las que escribe el programador encajan dentro de una jerarquía de clases. Digamos que usted escribe una clase que comienza con el encabezado siguiente:

```
public class Esfera
```

Si tal clase no tiene una superclase explícita, se considerará de manera implícita una subclase de la clase **Object**. Por lo tanto, toda clase es implícita o explícitamente una subclase de **Object** (por cierto, si desea hacer referencia a **Object** dentro de un programa, utilice la palabra clave **Object**. No olvide que todas las palabras clave de C# empiezan con minúscula).

En la Figura 11.2 se muestra un nuevo diagrama de clases; en él otra clase, llamada **Pelota**, también es una subclase de **Esfera**. Ahora el diagrama tiene una estructura tipo árbol cuya raíz, **Object**,



Figura 11.1 Diagrama de clases para las clases **Esfera** y **Burbuja**.

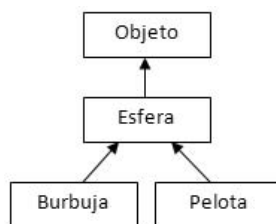


Figura 11.2 Diagrama de clases con estructura tipo árbol.

está en la parte superior. En general los diagramas de clases representan una jerarquía semejante a un árbol genealógico con un solo padre.

## ● La herencia en acción

---

Tal como vimos con `Burbuja`, es común que una clase tenga una superclase que, a su vez, cuenta con una superclase, y así sucesivamente, hasta llegar a la parte superior del árbol de herencia. En cualquier caso, no sólo se heredan los elementos `public` y `protected` de la superclase inmediata, sino también las variables, propiedades y métodos `public` y `protected` de todas las superclases del árbol. Esto es similar a la herencia genética: usted hereda las características de su madre, de su abuela, etcétera.

El lenguaje C# permite que una clase herede sólo de una superclase inmediata. A esto se le conoce como herencia simple. En la analogía familiar, esto significaría que usted puede heredar características de su madre, pero no de su padre.

Suponga que creamos un objeto `burbuja` a partir de la clase `Burbuja`:

```
Burbuja burbuja = new Burbuja();
```

¿Qué ocurre si utilizamos la propiedad `x` de la siguiente manera?

```
burbuja.X = 200;
```

Aquí `burbuja` es un objeto de la clase `Burbuja`, pero `x` es una propiedad de una clase distinta, llamada `Esfera`. La respuesta a la pregunta anterior es que todos los métodos y propiedades etiquetados como `public` (y `protected`) dentro de la superclase inmediata (y de todas las superclases en la jerarquía de clases) están disponibles para una subclase, y como `Burbuja` es una subclase de `Esfera`, `x` está disponible también para los objetos de la clase `Burbuja`.

Al utilizar un método o propiedad, la regla dicta que el sistema de C# primero busca en la descripción de la clase del objeto para tratar de encontrar el método. Si no lo encuentra ahí, busca en la descripción de clase de la superclase inmediata; si tampoco lo halla en ese lugar, examina la descripción de clase de la superclase de la superclase, y así recorre toda la jerarquía de clases hasta encontrar un método o propiedad con el nombre requerido. En la analogía familiar, diríamos que usted hereda implícitamente de su abuela, de su bisabuela, etcétera.

## ● base

---

En ocasiones es preciso que la clase invoque un método de su superclase inmediata, o de alguna de las clases del nivel superior en el árbol. No hay problema con esto, ya que los métodos que se hallan en todas las clases que ocupan los niveles superiores del árbol de herencia están disponibles para toda la “familia”, siempre y cuando estén etiquetados como `public` o `protected`. El único problema que puede surgir es cuando el método deseado se encuentra en la superclase y tiene el mismo nombre que un método de la clase actual (lo cual ocurre cuando se utiliza la redefinición). Para corregir este problema hay que anteponer al nombre del método la palabra clave `base`. Esto podría suceder, por ejemplo, para invocar el método `Mostrar` en una superclase:

```
base.Mostrar(áreaDibujo);
```



En general este sistema es más ordenado y corto que duplicar instrucciones, y nos puede ayudar a que nuestros programas sean más concisos, ya que aprovechan al máximo los métodos existentes.

## ● Constructores

Al analizar las clases en el capítulo 10, hablamos también sobre los constructores. Éstos nos permiten pasar parámetros a los objetos creados mediante la palabra clave **new**. Un constructor es un método con el mismo nombre que la clase. Recuerde que:

- si escribe una clase sin constructores, C# supone que sólo hay un constructor (con cero parámetros);
- si escribe una clase con uno o más constructores con parámetros y también necesita un constructor sin parámetros, debe escribirlo de manera explícita.

Los métodos constructores no se heredan. Esto significa que si necesita uno o varios constructores explícitos —como suele suceder— en una subclase, tendrá que escribirlos de manera manifiesta. Por ejemplo, suponga que tenemos una clase con dos constructores:

```
public class Globo
{
    protected int coordX, coordY, radio;

    public Globo()
    {
        coordX = 10;
        coordY = 10;
        radio = 20;
    }

    public Globo(int xInicial, int yInicial,
                int radioInicial)
    {
        coordX = xInicial;
        coordY = yInicial;
        radio = radioInicial;
    }

    // resto de la clase
}
```

Ahora imagine que queremos escribir una nueva clase llamada **GloboDiferente**, con la idea de que herede los atributos de la clase **Globo**; en ese caso, las opciones son:

1. No escribir un constructor. C# supondrá que hay un constructor sin parámetros.
2. Escribir uno o más constructores explícitos.
3. Escribir uno o más constructores explícitos que invoquen un constructor apropiado en la superclase, utilizando la palabra clave **base**.

Sin embargo, C# nos obliga a invocar un constructor en la superclase, tal como se describe a continuación. Si la primera instrucción en el constructor no es una invocación a un constructor en la superclase, C# invoca de manera automática al constructor sin parámetros que está en dicha superclase.

Para los programadores principiantes probablemente sea mejor dejar las cosas en claro y escribir de manera explícita una invocación a un constructor de la superclase, como se muestra en los siguientes ejemplos.

Veamos enseguida una subclase de `Globo` con un constructor que invoca el constructor sin parámetros de la superclase mediante el uso de `base`. El encabezado del constructor va seguido de un signo de dos puntos, después la palabra clave `base`, y por último los parámetros que se requieran:

```
public class GloboDiferente : Globo
{
    public GloboDiferente(int xInicial, int yInicial)
        : base()
    {
        coordX = xInicial;
        coordY = yInicial;
        radio = 20;
    }

    // resto de la clase
}
```

He aquí una subclase con un constructor que invoca de manera explícita el segundo constructor de la superclase, de nuevo utilizando la palabra clave `base`:

```
public class GloboModificado : Globo
{
    public GloboModificado (int xInicial, int yInicial,
                           int radioInicial)
        : base(xInicial, yInicial, radioInicial)
    {
    }

    // resto de la clase
}
```

## PRÁCTICAS DE AUTOEVALUACIÓN

**11.2** Escriba una nueva clase llamada `EsferaColoreada` que herede los atributos de la clase `Esfera` y proporcione un color que pueda establecerse al momento de crear el globo. Para ello utilice un método constructor, de manera que su clase permita escribir lo siguiente:

```
EsferaColoreada esferaColoreada =
    new EsferaColoreada(Color.Red);
```

**11.3** ¿Cuál es el problema con la subclase en el siguiente código?

```

public class CuentaBancaria
{
    protected int depósito;
    public CuentaBancaria(int depósitoInicial)
    {
        // resto del constructor
    }
    // resto de la clase
}

public class CuentaMejorada : CuentaBancaria
{
    public CuentaMejorada()
    {
        depósito = 1000;
    }
    // resto de la clase
}

```

## ● Clases abstractas

Considere un programa que mantiene formas gráficas de todos tipos y tamaños: círculos, rectángulos, cuadrados, triángulos, etc. Estas distintas formas, similares a las clases que hemos comentado en el capítulo, tienen información común: su posición, color y tamaño. A continuación declararemos una superclase llamada **Forma** que describa los datos comunes, y cada clase individual heredará esa información. La superclase se describe así:

```

public abstract class Forma
{
    protected int coordX, coordY ;
    protected int tamaño;
    protected Pen lápiz = new Pen(Color.Black);

    public void MoverDerecha()
    {
        coordX = coordX + 10;
    }

    public abstract void Mostrar(Graphics áreaDibujo);
}

```

Así, la clase para describir círculos hereda los atributos de la clase **Forma**, como se muestra a continuación:

```
public class Círculo : Forma
{
    public override void Mostrar(Graphics áreaDibujo)
    {
        áreaDibujo.DrawEllipse(lápiz, coordX, coordY, tamaño, tamaño);
    }
}
```

Sería inútil tratar de crear un objeto a partir de la clase **Forma**, ya que está incompleta. Ésa es la razón por la que se incluye la palabra clave **abstract** en el encabezado; el compilador evitará cualquier intento de crear una instancia a partir de esta clase. Se proporciona el método **MoverDerecha** completo, de manera que pueda ser heredado por cualquier subclase; no obstante, el método **Mostrar** es tan sólo un encabezado (sin cuerpo), y se describe con la palabra clave **abstract** para indicar que cualquier subclase debe proveer una implementación del mismo. En resumen, la clase **Forma** se denomina clase abstracta debido a que no existe como clase completa; su único propósito es ser usada en la herencia.

Hay una regla razonable que establece que si una clase contiene métodos o propiedades que sean **abstract**, también debe ser etiquetada como **abstract**.

Las clases abstractas nos permiten aprovechar las características comunes de las clases. Al declarar una clase como abstracta, el programador que la utiliza (mediante la herencia) se ve obligado a proveer los métodos faltantes. Por lo tanto, el diseñador de la clase puede fomentar un diseño en particular. Se emplea el término “abstracta” en razón de que, a medida que buscamos cada vez más alto en la jerarquía de clases, éstas se van haciendo gradualmente más generales o abstractas. En el ejemplo anterior la clase **Forma** es más abstracta (menos concreta) que la clase **Círculo**. La superclase abstrae las características (como la posición y el tamaño, en este ejemplo) que son comunes entre sus clases. Con frecuencia descubrimos en grandes programas orientados a objetos que los primeros niveles de las jerarquías de clases consisten en métodos abstractos. Esto es similar a lo que ocurre en biología, en donde la abstracción se utiliza para definir clases como la de los mamíferos, que no existen (por sí solas) pero sirven como superclases abstractas que nos permiten describir un diverso conjunto de subclases. Por ejemplo, nunca hemos visto un objeto mamífero, pero sí hemos visto una vaca, que es una instancia de una subclase de mamífero.

#### PRÁCTICA DE AUTOEVALUACIÓN

**11.4** Escriba una clase llamada **Cuadrado** que utilice la clase abstracta **Forma** antes descrita.

## Fundamentos de programación

Escribir programas como una colección de clases implica que éstos son modulares. Otro de los beneficios de este tipo de trabajo radica en que las partes de un programa pueden reutilizarse en otros programas. La herencia es una forma más en que la programación orientada a objetos provee el potencial de reutilización. Algunas veces los programadores se ven tentados a reinventar la rueda: desean escribir nuevo software cuando simplemente podrían utilizar el ya existente; quizá muestran esta acti-

tud porque programar resulta divertido, pero el problema radica en que al volverse cada vez más complejo, no hay suficiente tiempo para escribirlo desde cero. Imagine tener que escribir software para crear cada uno de los componentes de GUI que proporcionan las bibliotecas de C#; imagine tener que escribir una función matemática como `Sqrt` cada vez que se necesitara. Simplemente tomaría demasiado tiempo. En consecuencia, una buena razón para reutilizar software es que ahorramos no sólo el tiempo de escribirlo, sino también de probarlo exhaustivamente, lo cual suele ser una tarea todavía más absorbente. De ahí que la reutilización de clases tenga sentido.

Otra causa de que algunas veces los programadores no reutilicen el software, es que el existente no hace exactamente lo que necesitan. Tal vez haga noventa por ciento de lo que quieren, pero carece de ciertas características imprescindibles, o no hace las cosas como ellos desearían. Una metodología para enfrentar esta dificultad sería modificar el software existente para que coincida con las nuevas necesidades. Sin embargo, ésta es una estrategia peligrosa, equivalente a adentrarse en un campo minado: el software es bastante frágil; al tratar de modificarlo, se rompe. Esto se debe a que es muy fácil introducir errores nuevos y sutiles, los cuales requerirían un extenso proceso de depuración y corrección. Ésta es una experiencia tan común que los programadores se muestran muy reacios a modificar el software existente.

Aquí es donde la programación orientada a objetos entra en acción. En un programa orientado a objetos podemos heredar el comportamiento de aquellas partes de cierto software que necesitamos, redefinir los (pocos) métodos y/o propiedades que necesitamos que se comporten de manera distinta, y agregar nuevos métodos y/o propiedades que realicen acciones adicionales. A menudo podemos heredar la mayor parte de una clase y realizar sólo unos cuantos cambios necesarios. Únicamente tendríamos que probar las nuevas piezas, con la seguridad de que el resto ya ha sido probado con anticipación. El problema de la reutilización queda resuelto. Podemos utilizar con seguridad el software existente. Mientras tanto, la clase original permanece intacta, confiable y utilizable.

Una vez más, le invitamos a pensar que la herencia es como ir a comprar un automóvil: usted encuentra un modelo casi perfecto, pero le gustaría realizar unas pequeñas modificaciones (darle un color distinto o incluir un dispositivo de navegación por satélite). Para lograrlo podría “heredar” los atributos del automóvil estándar y modificar las partes que quisiera.

La programación orientada a objetos implica basarse en el trabajo de otros. El programador procede de esta forma:

1. Establece con claridad los requerimientos del programa.
2. Explora la biblioteca en busca de clases que realicen las funciones requeridas, y las usa para obtener los resultados deseados.
3. Revisa las clases empleadas en otros programas que haya escrito, y las utiliza según sea apropiado.
4. Amplía las clases de la biblioteca o sus propias clases mediante la herencia, siempre que esto sea posible.
5. Escribe sus propias clases nuevas.

Esto explica por qué los programas orientados a objetos suelen ser bastante compactos: simplemente utilizan las clases de biblioteca o crean nuevas clases que heredan de las clases de biblioteca. Esta metodología requiere una inversión de tiempo: el programador necesita un muy buen conocimiento de las bibliotecas. La idea de reutilizar software es tan poderosa que algunas personas sólo conciben la programación orientada a objetos, como el proceso de ampliar las clases de biblioteca de manera que se cumplan los requerimientos de una aplicación específica.

Casi todos los programas en este libro utilizan la herencia. Cada programa empieza con esta línea, que el entorno de desarrollo crea de manera automática:

```
public class Form1 : Form
```

En ella se indica que la clase `Form1` hereda características de la clase de biblioteca `Form`. Las características de `Form` incluyen métodos para crear una ventana de interfaz gráfica de usuario (GUI) con iconos para modificar su tamaño y cerrarla. Al introducir modificaciones a la clase `Form` para que responda a las necesidades de nuestros ejercicios estamos ampliando las clases de la biblioteca.

Tenga cuidado: a pesar de todas sus ventajas, en ocasiones la herencia no es la técnica apropiada. La composición (utilizar las clases existentes sin modificarlas) es una alternativa que puede ser más conveniente en ciertas circunstancias. En el capítulo 20, en donde hablaremos sobre diseño, analizaremos esta cuestión con más detalle.

### Errores comunes de programación

- Los programadores principiantes utilizan la herencia de una clase de biblioteca, la clase `Form`, desde su primer programa, pero aprender a utilizar la herencia dentro de nuestras propias clases lleva tiempo y requiere experiencia. Por lo general sólo vale la pena en programas de gran tamaño. No se preocupe si no utiliza la herencia durante un buen tiempo.
- Es común confundir la sobrecarga con la redefinición; sin embargo, recuerde:
  - *Sobrecargar* significa escribir dos o más métodos con el mismo nombre en la misma clase (pero con distintos parámetros).
  - *Redefinir* significa escribir un método en una subclase para utilizarlo en vez del método de la superclase (o de una de las superclases que están por encima de ella en el árbol de herencia).

### Nuevos elementos del lenguaje

- `:` – indica que esta clase hereda los atributos de otra clase.
- `protected` – describe como protegida una variable, propiedad o método, de manera que sólo puede utilizarse dentro de la clase o de cualquier subclase.
- `virtual` – describe un método o propiedad, estableciendo que puede redefinirse en una subclase.
- `override` – describe una propiedad o método que redefine un elemento en la superclase.
- `abstract` – describe como abstracta una clase, de manera que no se pueden crear objetos de esta clase, y se proporciona exclusivamente para ser heredada.
- `abstract` – describe como abstracta una propiedad o método dentro de una clase abstracta. Estas propiedades y métodos abstractos solo tienen el encabezado (sin cuerpo) y deben ser desarrollados dentro de la implementación de una subclase de la clase abstracta.
- `base` – el nombre de la superclase de una clase, la clase de la que hereda.
- `:` – va antes de la invocación a un constructor de la clase base.

### Resumen

- Ampliar (heredar) las herramientas de una clase es una buena forma de utilizar las partes existentes (clases) de otros programas.
- Una subclase hereda las herramientas de su superclase inmediata, y de todas las superclases que se encuentren por encima de ella en el árbol de herencia.
- Una clase sólo tiene una superclase inmediata (sólo puede heredar de una clase). A esto se le conoce como *herencia simple*, en la jerga de la programación orientada a objetos.

- Una clase puede ampliar las herramientas de una clase existente si proporciona uno o más de los siguientes elementos:
  - métodos y/o propiedades adicionales;
  - variables adicionales;
  - métodos y/o propiedades que redefinen (actúan en vez de) los métodos en la superclase.
- En términos de acceso, una variable, método o propiedad puede describirse como:
  - `public`: es decir, puede utilizarse desde cualquier clase.
  - `private`: puede emplearse sólo dentro de esta clase.
  - `protected`: puede usarse sólo dentro de esta clase y de cualquiera de sus subclases.
- Un diagrama de clases es un esquema ramificado que muestra las relaciones de herencia.
- Para hacer referencia al nombre de la superclase de una clase se utiliza la palabra clave `base`.
- Las clases abstractas se describen como `abstract`. No es posible instanciarlas para producir un objeto, ya que están incompletas, pero proveen variables, propiedades y métodos útiles, que las subclases pueden heredar.

## EJERCICIOS

- 11.1 Nave espacial** Escriba una clase llamada `NaveEspacial`, capaz de describir una aeronave ovalada con el mismo comportamiento que un objeto `Globo`. Trate de heredar lo más que pueda de las clases que analizamos en este capítulo.  
Dibuje un diagrama de clases para mostrar cómo se relacionan las distintas clases.
- 11.2 Billar** Escriba una clase llamada `BolaBillar`, de manera que restrinja los movimientos de una bola a los límites de un rectángulo que corresponda a las orillas acojinadas de una mesa de billar. Trate de heredar lo más que pueda de las clases que analizamos en este capítulo.
- 11.3 El banco** Una clase describe cuentas bancarias y provee los métodos `AbonarACuenta`, `RetirarDeCuenta` y `CalcularInterés`, y una propiedad `get` llamada `SaldoActual`. Para propósitos de este ejercicio debe haber dos tipos de cuentas: una regular y una dorada. Esta última produce un interés del 10%, mientras que la regular genera un interés del 1% en créditos mayores a \$100. Escriba clases que describan los dos tipos de cuenta, y utilice una clase abstracta para describir las características comunes (para simplificar, asuma que las cantidades de dinero se guardan como valores `int`).
- 11.4** Escriba una clase abstracta para describir objetos gráficos bidimensionales (cuadrado, círculo, rectángulo, triángulo, etc.) que tengan las siguientes características. Todos los objetos comparten variables `int` que especifican las coordenadas `x` y `y` de la esquina superior izquierda de un rectángulo circundante, además de variables `int` que describen la altura y el ancho del rectángulo. De igual manera, todos los objetos comparten las mismas propiedades `x` y `y` para establecer los valores de estas coordenadas, y las propiedades `Ancho` y `Alto` para establecer los valores correspondientes del objeto. Todos los objetos tienen una propiedad llamada `Área`, que devuelve el área del objeto, junto con un método `Mostrar` para mostrarla en pantalla, pero estos métodos son distintos dependiendo del objeto específico.

- 11.5** Un paquete de dibujo tridimensional soporta tres tipos de objetos: cubo, esfera y cono. Todos tienen una posición en el espacio, definida por las coordenadas  $x$ ,  $y$  y  $z$ . El tamaño de cada uno está definido por una altura, un ancho y una profundidad. Cada objeto proporciona un método para desplazar el objeto del origen a una posición en el espacio, y un método para girar alrededor de los ejes  $x$ ,  $y$  y  $z$ . Todos los objetos proporcionan un método para desplegarse en pantalla. Escriba una clase abstracta para describir estos objetos tridimensionales.

#### SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

---

**11.1**    `public class Bola : Esfera`  
      `{`  
          `public void MoverIzquierda(int cantidad)`  
          `{`  
              `coordX = coordX - cantidad;`  
          `}`  
          `public void MoverDerecha(int cantidad)`  
          `{`  
              `coordX = coordX + cantidad;`  
          `}`  
      `}`

**11.2**    `public class EsferaColoreada : Esfera`  
      `{`  
          `private Color color;`  
          `public EsferaColoreada(Color colorInicial)`  
          `{`  
              `color = colorInicial;`  
          `}`  
      `}`

- 11.3** El compilador encontrará una falla en la subclase. No hay una invocación explícita a un constructor en la superclase. Por lo tanto, C# tratará de invocar un constructor sin parámetros en la superclase, y no se ha escrito dicho método.

**11.4**    `public class Cuadrado : Forma`  
      `{`  
          `public override void Mostrar(Graphics áreaDibujo)`  
          `{`  
              `áreaDibujo.DrawRectangle(lápiz, coordX, coordY,`  
  `tamaño, tamaño);`  
          `}`  
      `}`





# Cálculos

**En este capítulo conocemos cómo:**

- desplegar números de la manera más conveniente;
- utilizar las funciones de la biblioteca matemática;
- llevar a cabo cálculos comerciales y científicos.

## ● Introducción

---

En el capítulo 4 vimos cómo llevar a cabo cálculos simples. En éste abordaremos los métodos para realizar cálculos más serios. Mejoraremos la explicación anterior y reuniremos toda la información necesaria para escribir programas que lleven a cabo cálculos. Si no le interesan los programas que realizan cálculos numéricos, puede pasar al siguiente capítulo.

Los cálculos surgen en muchos programas, no sólo en los que realizan cálculos matemáticos, científicos o de ingeniería. En los sistemas de información, la necesidad de realizar cálculos se da en la creación de nóminas, en los registros contables y en la generación de pronósticos. En los programas gráficos los cálculos son necesarios para escalar y desplazar imágenes en la pantalla.

En el capítulo 4 explicamos varias ideas importantes sobre los números y los cálculos. Tal vez quiera repasar ese capítulo antes de continuar. Los conceptos en cuestión son:

- entrada y salida de datos mediante el uso de cuadros de texto y etiquetas;
- conversión entre las representaciones de cadena de los números y sus representaciones internas;
- reglas de precedencia en las expresiones;
- conversiones en expresiones que mezclan datos `int` y `double`.

## ● Aplicación de formato a los números

---

Aplicar formato significa mostrar el texto de manera conveniente. La visualización de los números es muy importante, ya que debe ser lo más clara posible. Respecto de los números de punto flotante, por ejemplo, a veces resulta innecesario mostrar todos los decimales: si el valor `33.198765` representa el área de una habitación en metros cuadrados, tal vez sería suficiente desplegar la cifra `33`, sin posiciones decimales.

C# cuenta con diversas herramientas para aplicar formato a los valores; sin embargo, aquí nos limitaremos a comentar los casos más comunes, en los que se aplicará formato a valores `int` y `double`. La clase `String` tiene un método llamado `Format`, al cual podemos pasar dos parámetros:

- una cadena que contiene la información sobre el formato, y
- el valor al que debemos aplicar formato.

El método devuelve una cadena que contiene el valor con el formato, listo para ser mostrado (por ejemplo, en una etiqueta).

Empezaremos por ver cómo trabajar con los valores `int`. Suponga que tenemos el siguiente valor entero:

```
int i = 123;
```

Como sabemos, podemos emplear el método `ToString` para convertir un número:

```
label1.Text = Convert.ToString(i);
```

Con esto obtenemos la siguiente cadena de caracteres:

```
123
```

Pero ahora utilizaremos el método `Format` para aplicarle formato. Por ejemplo:

```
int i = 123;
label1.Text = String.Format("{0:D}", i);
```

Esto hace que la etiqueta reciba el siguiente valor:

```
123
```

Dentro de las llaves, el cero indica que hay sólo un número para darle formato; la `D` que está después del signo de dos puntos indica que la cadena se tratará como número decimal. El segundo parámetro es el número en sí (`i`, en este ejemplo).

En caso de saber anticipadamente que el entero puede tener, digamos, hasta cinco dígitos de longitud, y si quisiéramos alinear los números en formato tabular, podríamos utilizar una codificación como la siguiente:

```
int i = 123;
label1.Text = String.Format("{0:D5}", i);
```

con lo cual obtendríamos este resultado:

```
00123
```

Nuestro código aplica formato al número utilizando exactamente cinco dígitos alineados a la derecha, y completa con ceros las posiciones vacantes a la izquierda, según sea necesario.

Cuando tenemos números más grandes las comas pueden mejorar la legibilidad de la cifra. Esto se ilustra en el siguiente código:

```
int i = 123456;
label1.Text = String.Format("{0:N}", i);
```

Aquí **N** indica que el formato debe aplicarse a un número; el resultado es la siguiente cadena:

```
123,456.00
```

La aplicación de formato tiende a ser más útil cuando van a desplegarse valores `double`. En este ejemplo:

```
double d = 12.34;
label1.Text = String.Format("{0:F2}", d);
```

la etiqueta recibe el siguiente valor:

```
12.34
```

La **F** denota formato de punto flotante; el 2 especifica que se mostrarán dos dígitos después del punto decimal. A la izquierda del punto decimal se muestran los dígitos necesarios (por lo menos uno) para presentar todo el número. A la derecha del punto decimal el número se redondea para ajustarlo a los dos dígitos especificados.

C# provee también una herramienta para mostrar números de punto flotante en notación científica; en ese caso utilizamos la letra **E** dentro de la información de formato:

```
double número = 12000000;
label1.Text = String.Format("{0:E2}", número);
```

Mediante este código obtenemos lo siguiente:

```
1.20E+007
```

Por último, podemos mostrar los números en formato de moneda usando la letra **C** dentro de la cadena de formato. Por ejemplo, si empleamos este código:

```
double dinero = 12.34;
label1.Text = String.Format("{0:C}", dinero);
```

obtendremos:

```
$12.34
```

en donde el símbolo de moneda (\$) en este caso) se determina con base en la especificación local. El número se muestra siempre con dos dígitos después del punto decimal.

La tabla siguiente muestra un resumen de las opciones disponibles.

Tipo de datos	Ejemplo	Cadena con formato
decimal	<pre>int número = 123; label1.Text = String.Format("{0:D}", número);</pre>	123
decimal	<pre>int número = 123; label1.Text = String.Format("{0:D5}", número);</pre>	00123
punto flotante	<pre>double número = 12.34; label1.Text = String.Format("{0:F2}", número);</pre>	12.34
punto flotante	<pre>double número = 12.34; label1.Text = String.Format("{0:F0}", número);</pre>	12
científico	<pre>double número = 12000000; label1.Text = String.Format("{0:E2}", número);</pre>	1.20E+007
moneda	<pre>double dinero = 12.34; label1.Text = String.Format("{0:C}", dinero);</pre>	\$12.34
número	<pre>double número = 1234000; label1.Text = String.Format("{0:N2}", número);</pre>	1,234,000.00

### PRÁCTICA DE AUTOEVALUACIÓN

**12.1** Sabemos que el resultado de cierto cálculo será un número de punto flotante en el rango de 0 a 99, y queremos que aparezcan dos dígitos después del punto decimal. Seleccione un formato apropiado.

## ● Funciones y constantes de la biblioteca matemática

En los programas matemáticos, científicos o de ingeniería es común la utilización de funciones como seno, coseno y logaritmo. En C# estas funciones están incluidas en una de las bibliotecas: **Math**. Para usar una de las funciones que conforman dicha biblioteca podemos escribir algo como:

```
x = Math.Sqrt(y);
```

A continuación se listan algunas de las funciones de la biblioteca **Math** más utilizadas, en orden alfabético. Cuando el parámetro es un ángulo, debe expresarse en radianes.

<code>Abs(x)</code>	el valor absoluto de <code>x</code> , que algunas veces se escribe como $ x $ en matemáticas
<code>Ceiling(x)</code>	redondea <code>x</code> al valor <code>int</code> inmediato superior. También conocido como menor entero mayor o igual que <code>x</code>
<code>Cos(x)</code>	coseno del ángulo <code>x</code> , expresado en radianes
<code>Exp(x)</code>	$e^x$
<code>Floor(x)</code>	redondea <code>x</code> al valor <code>int</code> inmediato inferior. También conocido como mayor entero menor o igual que <code>x</code>
<code>Log(x)</code>	logaritmo natural de <code>x</code> (base $e$ )
<code>Log10(x)</code>	logaritmo base 10 de <code>x</code>
<code>Max(x, y)</code>	valor máximo entre <code>x</code> y <code>y</code>
<code>Min(x, y)</code>	valor mínimo entre <code>x</code> y <code>y</code>
<code>Pow(x, y)</code>	<code>x</code> elevado a la potencia <code>y</code> , o $x^y$
<code>Round(x)</code>	redondea un valor <code>double</code> al valor <code>int</code> más cercano. Por ejemplo, <code>Math.Round(3.4)</code> es 3.0, con un punto decimal.
<code>Sin(x)</code>	seno del ángulo <code>x</code> , expresado en radianes
<code>Sqrt(x)</code>	la raíz cuadrada positiva de <code>x</code>
<code>Tan(x)</code>	la tangente del ángulo <code>x</code> , expresado en radianes

Al utilizar estos métodos, algunas veces debemos tener cuidado en cuanto al tipo de variables o literales que empleamos como parámetros. Por ejemplo, el método `Abs` puede recibir cualquier valor numérico, pero el método `Cos` sólo puede recibir un número `double`.

Las constantes matemáticas pi ( $\pi$ ) y  $e$  también están disponibles en la biblioteca `Math`; para trabajar con ellas podemos escribir:

```
double x, y;
x = Math.PI;
y = Math.E;
```

## ● Constantes

Las constantes son valores que no cambian mientras el programa se ejecuta. En la sección anterior hablamos de dos de estas constantes: `E` y `PI`, incluidas en la clase `Math`. Pero también es muy frecuente encontrarnos con otros valores que no cambiarán a lo largo de la ejecución de un programa. Algunos ejemplos podrían ser el factor para convertir pulgadas en centímetros, o la velocidad de la luz. Una manera para utilizarlos consiste en escribir los valores para estas cantidades directamente en el programa, como en el siguiente ejemplo:

```
cm = pulgadas * 2.54;
```

Una forma más conveniente de hacerlo es declarar dichos números como variables con un valor constante, y darles un nombre, por ejemplo:

```
const double pulgACm = 2.54;
```

Las variables que se declaran de esta manera no pueden modificarse (mediante una asignación, por ejemplo) cuando el programa se ejecuta. De hecho el compilador rechazará cualquier intento por darles un nuevo valor. Entonces, podemos usar el nombre en el siguiente cálculo:

```
cm = pulgadas * pulgACm;
```

Esta instrucción es más clara, ya que utilizamos el lenguaje de programación para explicar lo que estamos haciendo, en vez de usar números que por sí solos no sabemos lo que significan.

### PRÁCTICA DE AUTOEVALUACIÓN

**12.2** La velocidad de la luz es de 299,792,458 metros por segundo. Escriba este valor como una constante `double`.

### ● Ejemplo práctico: dinero

A continuación analizaremos el desarrollo de un programa en el que se realizan cálculos numéricos con dinero. En casi todos los países la notación monetaria consta de dos partes: dólares y centavos, pesos y centavos, euros y centavos, libras y peniques. Esto nos da la opción de representar un monto de dinero ya sea como cantidad `double` (20.25 dólares) o como `int` (2025 centavos). Si utilizamos centavos tendremos que convertir las cantidades en dólares y centavos, y viceversa. En este caso emplearemos variables `double` para representar valores.

Construyamos ahora un programa que calcule el interés compuesto. Digamos que se invierte cierta cantidad de dinero a una tasa de interés anual específica, y los rendimientos se suman a su valor. El usuario introduce en cuadros de texto la cantidad inicial invertida (como número entero) y una tasa de interés (una cifra que puede incluir decimales). Después hace clic en un botón para ver el monto acumulado por año, como se muestra en la Figura 12.1.

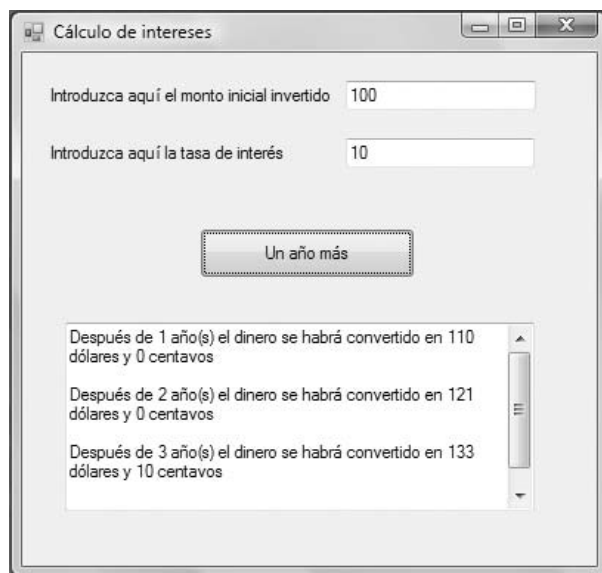


Figura 12.1 El programa Cálculo de intereses.

Primero declararemos las cantidades principales:

```
double tasa, montoNuevo;
double montoAnterior;
```

Al hacer clic en el botón para calcular el interés acumulado en el siguiente año, el programa debe realizar estas operaciones:

```
montoNuevo = montoAnterior + (montoAnterior * tasa / 100);
```

Como ya mencionamos, al desplegar cantidades de dinero es preciso mostrar un número entero (dólares, euros, pesos...) y un número entero de centavos; por ejemplo, si el valor es 127.2341, lo más conveniente sería mostrarlo como 127 dólares y 23 centavos.

Para comenzar veamos cómo trabajar con el número entero que representa los dólares. Si utilizamos el operador de conversión (`int`) podemos convertir el número `double` en un `int`, truncando la parte fraccionaria:

```
dólares = (int) montoNuevo;
```

Veamos ahora lo que concierne a los centavos. Antes que nada necesitamos deshacernos de la parte del número correspondiente a los dólares. Para ello podemos restar el número entero de dólares, de manera que un número como 127.2341 se convertiría en 0.2341. Luego multiplicamos ese número por 100.0 para convertirlo en centavos, de manera que 0.2341 se transformaría en 23.41. A continuación usamos `Math.Round` para convertir esa cantidad al número entero más cercano (23.0). Por último, convertimos el valor `double` en un valor `int` mediante el uso de (`int`).

```
centavos = (int) Math.Round(100 * (montoNuevo - dólares));
```

Una vez realizados los cálculos anteriores estamos en posibilidad de desplegar los valores convertidos apropiadamente. Para ello tenemos que:

```
montoAnterior = montoNuevo;
```

y esto es de lo que tratan las inversiones.

El siguiente es el código completo del programa; el formulario aparece en la Figura 12.1. A nivel de clase, las declaraciones de variables de instancia son:

```
private int año = 1;
private double montoAnterior;
```

La respuesta a un clic del botón es:

```
private void button1_Click(object sender, EventArgs e)
{
    double tasa, montoNuevo;
    int dólares, centavos;

    if (año == 1)
    {
        montoAnterior = Convert.ToDouble(textBox1.Text);
    }

    rate = Convert.ToDouble(textBox2.Text);
```

```

montoNuevo = montoAnterior + (montoAnterior * tasa / 100);

dólares = (int)montoNuevo;
centavos = (int)Math.Round(100 * (montoNuevo - dólares));
textBox3.AppendText("Después de " + Convert.ToString(año)
    + " año(s) "
    + "el dinero se habrá convertido en "
    + Convert.ToString(dólares) + " dólares y "
    + Convert.ToString(centavos) + " centavos" + "\r\n\r\n");
montoAnterior = montoNuevo;
año++;
}

```

### ● Ejemplo práctico: iteración

En la programación que involucra números es muy común escribir repeticiones o iteraciones, es decir, ciclos que buscan la solución a determinada ecuación hasta encontrarla con una precisión adecuada.

Como ejemplo del uso de repeticiones, veamos una fórmula para obtener el seno de un ángulo:

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$$

(Tenga en cuenta que si fuera necesario conocer el seno de un ángulo en un programa no tendríamos que escribir explícitamente esta fórmula, ya que está disponible como función de biblioteca).

Como podemos ver, cada término se deriva del anterior, con base en la siguiente multiplicación:

$$-x^2/((n + 1) \times (n + 2))$$

Por lo tanto, es posible construir un ciclo que se repita hasta que el nuevo término encontrado sea menor que cierta cifra aceptable, digamos, 0.0001.

```

private double Sin(double x)
{
    double término, resultado;

    resultado = 0.0;
    término = x;
    for ( int n = 1; Math.Abs(término) >= 0.0001; n = n + 2 )
    {
        resultado = resultado + término;
        término = - término * x * x / ((n + 1) * (n + 2));
    }
    return resultado;
}

```

En este código el método de biblioteca `Abs` calcula el valor absoluto de su parámetro.



## ● Gráficas

Es bastante usual que la información matemática, de ingeniería y financiera se presente de manera gráfica. Por ello, en nuestro siguiente ejercicio veremos un programa para graficar funciones matemáticas. Suponga que queremos trazar esta función:

$$y = ax^3 + bx^2 + cx + d$$

en donde los valores para  $a$ ,  $b$ ,  $c$  y  $d$  se introducen mediante barras de seguimiento, como se muestra en la Figura 12.2.

Antes de construir este programa debemos resolver varias cuestiones de diseño. En primer lugar, queremos ver la gráfica de manera que los valores positivos de la coordenada  $y$  aumenten hacia arriba, mientras que las coordenadas de los píxeles  $y$  se miden hacia abajo. En consecuencia, tendremos que distinguir entre  $x$  y su coordenada de píxel equivalente, `pixelX`, y entre  $y$  y `pixelY`.

Además debemos asegurarnos de que la gráfica se ajuste de manera conveniente al cuadro de imagen; es decir, que no sea demasiado pequeña ni demasiado grande, para lo cual tendremos que realizar un escalado apropiado. Supongamos que el área disponible en el cuadro de imagen es de 200 píxeles en la dirección  $x$ , y de 200 píxeles en la dirección  $y$ . Dado que diseñaremos el programa de manera que se muestren los valores  $x$  y  $y$  en el rango de  $-5.0$  a  $+5.0$ , una unidad de  $x$  (o de  $y$ ) equivale a 20 píxeles.

Por último, ya que utilizaremos el método `DrawLine` para trazar la gráfica, tendremos que dibujar una forma curvada a partir de un gran número de líneas pequeñas. Para ello nos desplazaremos

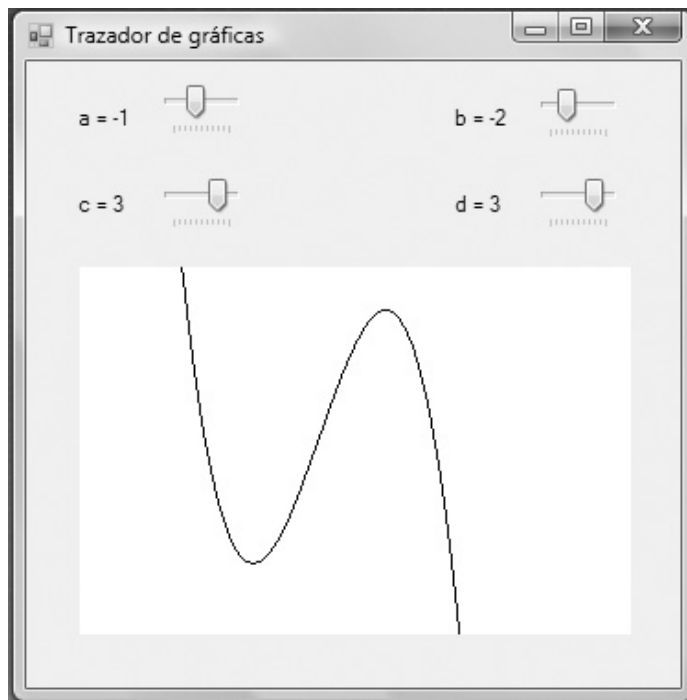


Figura 12.2 El programa Trazador de gráficas, desplegando la gráfica correspondiente a una función cúbica.

a lo largo de la dirección  $x$ , un píxel a la vez, trazando una línea desde la coordenada  $y$  equivalente hasta la siguiente. Por cada píxel  $x$ , el programa debe calcular:

1. el valor de  $x$  a partir del valor del píxel  $x$ ,
2. el valor de  $y$ , que es el valor de la función,
3. el valor del píxel  $y$  a partir del valor de  $y$ ,

para lo cual utilizamos las siguientes instrucciones:

```
x = EscalarX(píxelX);  
y = LaFunción(x);  
píxelY = EscalarY(y);
```

Después el programa avanza al siguiente píxel  $x$  y calcula de nuevo el píxel  $y$  equivalente:

```
siguientePíxelX = píxelX + 1;  
siguienteX = EscalarX(sigüientePíxelX);  
sigüienteY = LaFunción(sigüienteX);  
sigüientePíxelY = EscalarY(SigüienteY);
```

Por último se traza la pequeña sección de la curva:

```
papel.DrawLine(lápiz, píxelX, píxelY, sigüientePíxelX, sigüientePíxelY);
```

Cabe mencionar que el programa utiliza varios métodos privados para ayudarnos a simplificar la lógica. Además sólo se emplea un método para manejar los eventos de las cuatro barras de seguimiento. He aquí el código completo para este programa de trazado de gráficas.

A nivel de clase, las variables son:

```
private double a, b, c, d;  
private Graphics papel;  
private Pen lápiz = new Pen(Color.Black);
```

Al momento de mover la primera barra de seguimiento, el evento se maneja mediante el siguiente método:

```
private void trackBarA_Scroll(object sender, EventArgs e)  
{  
    TrazarGráfica();  
}
```

Lo mismo aplica con las demás barras de seguimiento. A su vez, los métodos de manejo de eventos invocan los siguientes métodos:

```
private void TrazarGráfica()  
{  
    papel = pictureBox1.CreateGraphics();  
  
    a = trackBarA.Value;  
    labelA.Text = "a = " + Convert.ToString(a);  
    b = trackBarB.Value;  
    labelB.Text = "b = " + Convert.ToString(b);  
    c = trackBarC.Value;
```

```

        labelC.Text = "c = " + Convert.ToString(c);
        d = trackBarD.Value;
        labelD.Text = "d = " + Convert.ToString(d);
        paper.Clear(Color.White);
        Trazar();
    }

    private void Trazar()
    {
        double x, y, siguienteX, siguienteY;
        int píxelX, píxelY, siguientePíxelX, siguientePíxelY;
        for (píxelX = 0; píxelX <= pictureBox1.Width; píxelX++)
        {
            x = EscalarX(píxelX);
            y = LaFunción(x);
            píxelY = EscalarY(y);
            siguientePíxelX = píxelX + 1;
            siguienteX = EscalarX(siguientePíxelX);
            siguienteY = LaFunción(siguienteX);
            siguientePíxelY = EscalarY(siguienteY);
            papel.DrawLine(lápiz, píxelX, píxelY,
                           siguientePíxelX, siguientePíxelY);
        }
    }

    private double LaFunción(double x)
    {
        return a * x * x * x + b * x * x + c * x + d;
    }

    private double EscalarX(int píxelX)
    {
        double xInicial = -5, xFinal = 5;
        double xEscala = pictureBox1.Width / (xFinal - xInicial);
        return (píxelX - (pictureBox1.Width / 2)) / xEscala;
    }

    private int EscalarY(double y)
    {
        double yInicial = -5, yFinal = 5;
        int coordPíxel;
        double yEscala = pictureBox1.Height / (yFinal - yInicial);

        coordPíxel = (int) (-y * yEscala) +
                     (int) (pictureBox1.Height / 2);
        return coordPíxel;
    }
}

```

Ejecute este programa y modifique los valores de las barras de seguimiento para ver el efecto del cambio de parámetros. También podemos trazar ecuaciones cuadráticas (si hacemos el valor del coeficiente *a* igual a cero) y líneas rectas.

## ● Excepciones

---

Si está leyendo este capítulo por primera vez, probablemente sea mejor que omita esta sección, ya que trata —como indica el título— sobre eventos que no ocurren con mucha frecuencia.

Al escribir un programa que realiza cálculos debemos tener cuidado de no exceder el tamaño de los números permitidos. No es como realizar un cálculo en una hoja de papel, en donde los números pueden ser del tamaño que queramos; es algo más parecido a utilizar una calculadora con un límite superior finito en cuanto al tamaño de los números que puede contener.

Por ejemplo, si declara un entero de la siguiente manera:

```
int número;
```

debe tener en cuenta que el número más grande que se puede guardar en una variable `int` es extenso, pero se limita a 2,147,483,647. Por lo tanto, si escribe:

```
número = 2147483647;
número = número + 2147483647;
```

el resultado de la suma no podrá guardarse como valor `int`. El programa terminará y aparecerá un mensaje de error. A esto se le conoce como *desbordamiento*, y es una de las posibles *excepciones* que pueden surgir al momento de ejecutar un programa.

Es posible que el desbordamiento se presente con más sutileza que en el ejemplo anterior, especialmente cuando un usuario introduce datos en un cuadro de texto y, en consecuencia, su tamaño es impredecible. Por ejemplo, he aquí un programa simple para calcular el área de una habitación, en donde podría ocurrir un desbordamiento:

```
int longitud, área;
longitud = Convert.ToInt32(entradaTextBox.Text);
área = longitud * longitud;
```

Algunas situaciones que podrían provocar un desbordamiento son:

- sumar dos números grandes;
- restar un número positivo grande a un número negativo grande;
- dividir entre un número muy pequeño;
- multiplicar dos números grandes.

De todo esto podemos concluir que aun cuando un simple cálculo parezca inofensivo, necesitamos vigilarlo. Hay varias formas de lidiar con una excepción:

1. Ignorarla, esperando que no ocurra, y estar preparados para que el programa falle y/o genere resultados extraños cuando se produzca la excepción. Esto está bien para los programadores principiantes, pero es absolutamente inconveniente en el caso de los programas reales, cuyo diseño debe ser robusto.
2. Permitir que surja la excepción, pero darle solución escribiendo un manejador de excepciones, como veremos en el capítulo 17.

3. Evitarla escribiendo comprobaciones que garanticen la imposibilidad de que se den situaciones semejantes. Por ejemplo, en un programa para calcular el área de una habitación podemos evitar el desbordamiento si comprobamos el tamaño de los datos:

```
if (longitud > 10000)
{
    respuestaTextBox.Text = "valor demasiado grande";
}
```

Ya vimos cómo puede ocurrir un desbordamiento cuando un programa utiliza valores `int`. Podríamos esperar problemas similares si los valores `double` se volvieran demasiado grandes, pero no es así. Si un valor de este tipo se vuelve demasiado grande, el programa sigue funcionando y el valor toma uno de los valores especiales: `NaN` (no es un número), `PositiveInfinity` (infinito positivo) o `NegativeInfinity` (infinito negativo), según sea apropiado.

## Fundamentos de programación

- Casi todos los programas científicos, de ingeniería, matemáticos y estadísticos emplean numerosos cálculos, e incluso los programas pequeños que podría parecer que no los necesitan suelen utilizar algo de aritmética.
- El primer paso (y el más importante) al trabajar con cálculos consiste en decidir qué tipos de variables utilizar para representar los datos. La principal elección es entre `int` y `double`.
- Es común utilizar repeticiones o iteraciones en los cálculos numéricos cuya solución converge hacia la respuesta. Para ello se requiere el uso de ciclos.
- La biblioteca de funciones matemáticas es invaluable en los programas de este tipo.
- Durante la realización de cálculos es posible que se presenten situaciones excepcionales, como el desbordamiento, y debemos anticiparnos a ellas si queremos que nuestro programa sea robusto en toda circunstancia.

## Errores comunes de programación

- Las situaciones excepcionales, como tratar de dividir entre cero, son capaces de producir resultados extraños o el término de la ejecución del programa. Haga que sus programas sean robustos.

## Resumen

- Las principales formas de representar números son `int` o `double`. Estos tipos de datos producen distintas precisiones y rangos.
- Las variables pueden declararse como `const`. En ese caso es imposible modificar su valor cuando el programa está en ejecución.
- Las funciones de biblioteca proveen las funciones matemáticas más comunes; por ejemplo, la necesaria para obtener el seno de un ángulo.
- El programador debe tener en cuenta las excepciones que podrían surgir durante la realización de cálculos.

## EJERCICIOS

- 12.1 Costo de una llamada telefónica** Una llamada telefónica cuesta 10 centavos por minuto. Escriba un programa que reciba mediante cuadros de texto la duración de una llamada telefónica, expresada en horas, minutos y segundos, y muestre el costo de la misma en centavos.
- 12.2 Conversión de medidas** Escriba un programa que reciba como entrada una medida, expresada en pies y pulgadas, mediante dos cuadros de texto, y la convierta a centímetros (hay 12 pulgadas en un pie; una pulgada equivale a 2.54 centímetros).
- 12.3 Clic del ratón** Escriba un programa que despliegue un cuadro de imagen. Cuando el usuario haga clic en él deberá aparecer un cuadro de mensaje que indique la distancia que hay entre la posición del cursor y la esquina superior izquierda del cuadro.
- 12.4 Caja registradora** Escriba un programa que represente una caja registradora. Los montos de dinero pueden introducirse en un cuadro de texto y se suman automáticamente al total acumulado; éste debe desplegarse en otro cuadro de texto. Es necesario que haya un botón que permita borrar la suma (hacerla cero).
- 12.5 Suma de enteros** La suma de los enteros de 1 a  $n$  se obtiene mediante la siguiente fórmula:

$$\text{suma} = n(n + 1)/2$$

Escriba un programa que reciba como entrada un valor para  $n$  mediante un cuadro de texto, y que calcule la suma de dos formas: primero utilizando la fórmula y después sumando los números a través del uso de un ciclo.

- 12.6 Números aleatorios** Los números aleatorios suelen utilizarse en los programas computacionales y de simulación, conocidos como métodos de Montecarlo. En secciones anteriores de este libro comentamos cómo utilizar la biblioteca de clase `Random`, que nos permite crear un generador de números aleatorios como se muestra a continuación:

```
Random generador = new Random();
```

Esta clase tiene un método llamado `Next`, el cual devuelve un número aleatorio como valor `int` en cualquier rango que elijamos (mismo que se especifica mediante parámetros). Por ejemplo:

```
int número;
número = generador.Next(1, 6);
```

crea un número aleatorio en el rango de 1 a 6.

Escriba un programa para verificar el método generador de números aleatorios al pedirle 100 números aleatorios que tengan el valor 1 o 2. Cuente y muestre el número de valores iguales a 1 y el número de valores iguales a 2. Proporcione un botón que permita crear otro conjunto de 100 valores.

- 12.7 Series para e** El valor de  $e^x$  puede calcularse mediante la suma de la siguiente serie:

$$e^x = 1 + x + x^2/2! + x^3/3! + \dots$$

Escriba un programa que reciba como entrada un valor de  $x$  mediante un cuadro de texto, y que calcule  $e^x$  hasta cierto grado de precisión deseado. Compruebe el valor comparándolo con el que se obtiene al hacer referencia a la constante `E` de la biblioteca `Math`.

- 12.8 Cálculo de impuestos** Escriba un programa que lleve a cabo el cálculo de impuestos. Los primeros \$10,000 están exentos de pago, pero la tasa es de 33% para cualquier monto mayor a ése. Escriba el programa para recibir como entrada un salario en dólares mediante un cuadro de texto y calcular los impuestos a pagar. Vigile los errores al realizar el cálculo; ¡la respuesta tiene que ser precisa hasta el centavo más cercano!

- 12.9 Área de un triángulo** La fórmula para calcular el área de un triángulo cuyos lados tienen una longitud  $a$ ,  $b$  y  $c$  es:

$$\text{área} = \sqrt{s(s-a)(s-b)(s-c)}$$

en donde:

$$s = (a + b + c)/2$$

Escriba un programa que reciba como entrada los tres valores de los lados de un triángulo mediante cuadros de texto, y utilice esta fórmula para calcular el área. Su programa debe comprobar primero que las tres longitudes especificadas en realidad formen un triángulo. Por ejemplo,  $a + b$  debe ser mayor que  $c$ .

- 12.10 Raíz cuadrada** La raíz cuadrada de un número puede calcularse mediante iteraciones, como se muestra a continuación. Escriba un programa para hacerlo con un número que se introduzca mediante un cuadro de texto.

- La primera aproximación a la raíz cuadrada de  $x$  es  $x/2$ .
- Las aproximaciones sucesivas se obtienen mediante la fórmula:

$$\text{siguienteAproximación} = (\text{últimaAproximación}^2 - x)/2 + \text{últimaAproximación}$$

Compare el valor con el que se obtiene al utilizar el método de biblioteca `sqrt`.

- 12.11 Calculadora matemática** Escriba un programa que actúe como una calculadora matemática. Ésta deberá tener botones para introducir números, y éstos tendrán que desplegarse como en la pantalla de una calculadora de escritorio. También es preciso que haya botones para realizar cálculos matemáticos estándar, como seno, coseno, logaritmo natural y raíz cuadrada.

- 12.12 Calculadora de interés** Modifique la parte del cálculo del programa anterior, de manera que se pueda utilizar un número `int` (en vez de `double`) para representar un monto de dinero (expresado en centavos).

- 12.13 Graficador** Mejore el programa para trazar gráficas que vimos en este capítulo, de manera que:

- trace los ejes  $x$  y  $y$ ;
- reciba como entrada los coeficientes mediante cuadros de texto en vez de barras de seguimiento (para tener precisión);
- reciba como entrada factores de escala horizontal y vertical (acercamiento) mediante barras de seguimiento;

- trace una segunda gráfica de la misma función, pero con coeficientes distintos;
- trace las gráficas de algunas otras funciones. Una manera de hacerlo sería volver a codificar el método `LaFunción`.

**12.14 Integración numérica** Escriba un programa que calcule la integral de una función y utilizando la “regla del trapecio”. El área que se ubica bajo la gráfica de la función se divide en  $n$  bandas iguales, de ancho  $d$ ; el área que está bajo la curva (la integral) es aproximadamente igual a la suma de todos los (pequeños) trapecios:

$$\text{área} \cong d(y_0 + 2y_1 + 2y_2 + \dots + 2y_{n-1} + y_n)/2$$

o:

área = (la mitad del ancho de la banda)  $\times$  (primera + última + dos veces la suma de las demás)

Utilice una función cuya respuesta conozca, y experimente empleando valores cada vez más pequeños de  $d$ .

**12.15 Conjunto de Mandelbrot** El conjunto de Mandelbrot (Figura 12.3) es una famosa e impactante imagen que se produce al evaluar repetitivamente una fórmula en cada punto, en un espacio de dos dimensiones. Tome un punto con las coordenadas  $x_{\text{inicial}}$  y  $y_{\text{inicial}}$ . Después calcule de manera repetitiva nuevos valores de  $x$  y de  $y$  a partir de los valores anteriores, mediante el uso de las siguientes fórmulas:

$$x_{\text{nueva}} = x_{\text{anterior}}^2 - y_{\text{anterior}}^2 - x_{\text{inicial}}$$

$$y_{\text{nueva}} = 2x_{\text{anterior}}y_{\text{anterior}} - y_{\text{inicial}}$$

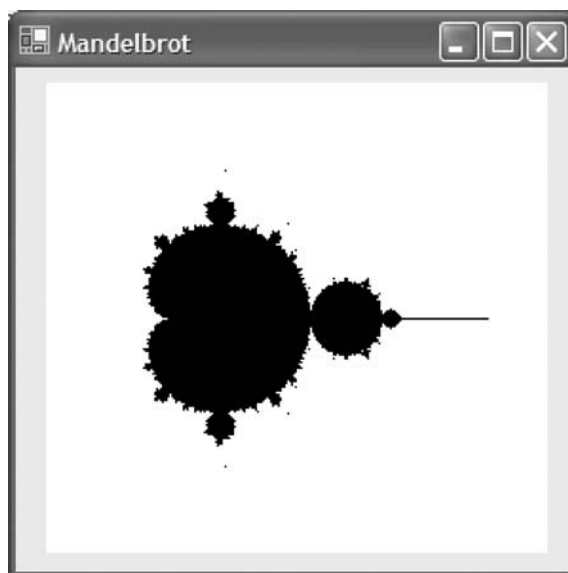


Figura 12.3 El conjunto de Mandelbrot.



Los primeros valores de  $x_{\text{anterior}}$  y  $y_{\text{anterior}}$  son  $x_{\text{inicial}}$  y  $y_{\text{inicial}}$ . Para cada iteración, calcule  $r = \sqrt{x_{\text{nueva}}^2 + y_{\text{nueva}}^2}$ . Repita hasta que  $r > 10,000$  o hasta llegar a 100 iteraciones, lo que ocurra primero. Si  $r$  es mayor que 10,000, coloree de blanco el píxel correspondiente a esta coordenada; en caso contrario coloréelo de negro.

Repita el proceso para todos los puntos con un valor de  $x$  entre  $-1.0$  y  $+2.0$ , y con un valor de  $y$  en el rango de  $-2.0$  a  $+2.0$ .

A medida que avanza la iteración, partiendo de valores específicos de  $x_{\text{inicial}}$  y  $y_{\text{inicial}}$ , el valor de  $r$  algunas veces permanece razonablemente pequeño (cerca de 1.0). Para otros valores de  $x_{\text{inicial}}$  y  $y_{\text{inicial}}$  el valor de  $r$  se vuelve rápidamente muy grande y tiende a dispararse hacia el infinito.

## SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

```
12.1 double respuesta;
    label1.Text = String.Format("{0:N2}", respuesta);
```

```
12.2 const double velocidadLuz = 299792458;
```

# Estructuras de datos: cuadros de lista y listas

## En este capítulo conoceremos:

- cómo utilizar el control cuadro de lista;
- qué es una lista;
- cómo agregar, insertar y eliminar elementos en un cuadro de lista;
- cómo obtener la longitud de un cuadro de lista;
- qué es un índice;
- cómo llevar a cabo operaciones comunes en un cuadro de lista, como búsquedas rápidas, adición de elementos y búsquedas detalladas;
- cómo utilizar listas genéricas.

## ● Introducción

---

Los controles cuadro de lista (ListBox) muestran un listado de cadenas de caracteres dentro de un cuadro. Además proporcionan varias capacidades, incluyendo la habilidad de seleccionar un elemento de la lista al hacer clic en él, agregar elementos y eliminarlos. El control cuadro de lista está disponible junto con los demás en el cuadro de herramientas, y puede colocarse de la misma manera que cualquiera de los controles en un formulario. Para ver su funcionamiento utilizaremos como ejemplo una lista de compras a la que agregaremos elementos uno por uno. En la Figura 13.1 se muestra ese cuadro de lista después de agregar varios elementos, cada uno de los cuales ocupa una sola línea. Si no se puede mostrar la lista completa en el espacio disponible, aparecerá de manera automática una barra de desplazamiento. Más adelante veremos cómo eliminar elementos de la lista.

Los cuadros de lista constituyen una buena introducción al uso de estructuras de datos, ya que proveen una representación visual directa de la información. En este capítulo exploraremos su utilización como estructuras de datos, así que puede leerlo y estudiar su contenido de manera independiente a los capítulos sobre arreglos.



Figura 13.1 Un cuadro de lista.

## ● Listas

Cuando arrastramos un cuadro de lista al formulario desde el cuadro de herramientas, estamos creando una nueva instancia de la clase `ListBox`. Esta clase emplea otra clase llamada `List` para llevar a cabo sus funciones. Los cuadros de lista simplemente *despliegan* la información en el formulario, y manejan los eventos de clic de ratón; por su parte, las listas *almacenan* la información que se muestra en aquellos. Así, mientras los cuadros de lista soportan los eventos `Click` y `DoubleClick` junto con propiedades tales como `SelectedItem`, las listas proveen métodos para agregar y eliminar elementos de éstas.

Por ejemplo, suponga que creamos un cuadro de lista llamado `compras`. En ese caso la propiedad `compras.Items` es un objeto —la lista propiamente dicha— que contiene la información mostrada en el cuadro. Una vez comprendido lo anterior, estamos preparados para usar las propiedades y métodos de las listas. Por ejemplo, podemos obtener un conteo del número de elementos que conforman la lista (y el cuadro de lista) mediante el uso de la propiedad `Count`, como se muestra a continuación:

```
int númeroDeElementos = compras.Items.Count;
```

## ● Adición de elementos a una lista

El programa de ejemplo que se muestra en la Figura 13.2 permite que el usuario agregue elementos a un cuadro de lista.

Este método responde al clic del botón y coloca un elemento a comprar al final del cuadro de lista.

```
private void button1_Click(object sender, EventArgs e)
{
    compras.Items.Add(textBox1.Text);
}
```

En este ejemplo el nombre del cuadro de lista es `compras`. Como vimos antes, una de las propiedades de los cuadros de lista es `Items`, y representa su contenido como una instancia de la clase `List`. A su

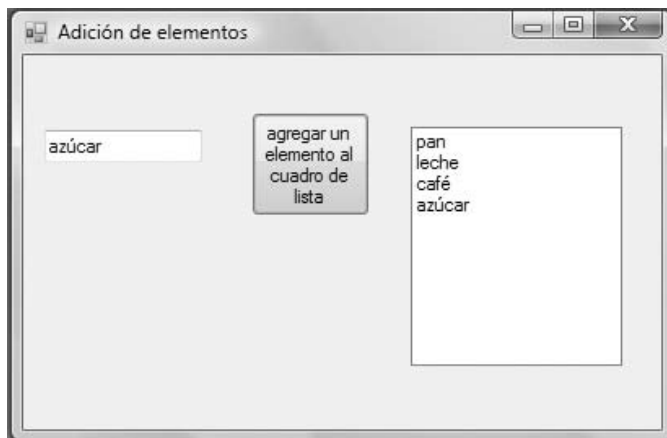


Figura 13.2 Adición de elementos a una lista de compras.

vez, esta clase proporciona varios métodos, uno de los cuales es `Add`, que nos permite añadir elementos a una lista. Su parámetro es el valor que se agregará a la lista.

También podemos colocar elementos en un cuadro de lista en tiempo de diseño. Al seleccionar la propiedad `Items` de un cuadro de lista aparece una nueva ventana que nos permite insertar elementos en él.

### ● La longitud de una lista

A continuación veremos un método que responde al clic de un botón desplegando un cuadro de mensaje con el número de elementos que contiene el cuadro de lista.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(Convert.ToString(compras.Items.Count));
}
```

Una vez más podemos ver cómo se utiliza la propiedad `Items` del cuadro de lista llamado `compras`. Al mismo tiempo se emplea la propiedad `Count` de la clase `List` para obtener el número de elementos que lo conforman.

### ● Índices

Los programas utilizan un *índice* para hacer referencia a los elementos que constituyen un cuadro de lista. Un índice es un entero que indica a qué elemento se hace referencia. El primer elemento tiene el índice 0, el segundo 1, etcétera. Podemos comparar este cuadro de lista con una tabla simi-

0	pan
1	leche
2	café

Figura 13.3 Diagrama de un cuadro de lista, mostrando sus índices.

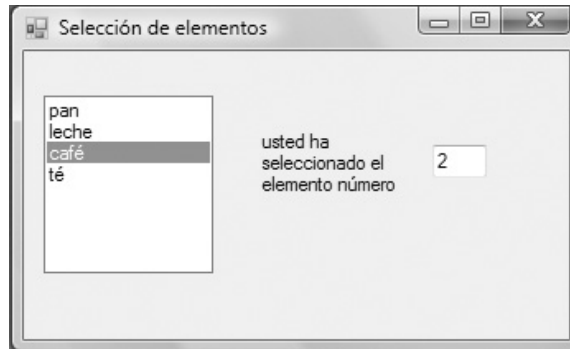


Figura 13.4 Selección de elementos en un cuadro de lista.

lar a la que se muestra en la Figura 13.3, con los valores de los índices a un lado (estos valores no se guardan junto con los datos).

Ahora analicemos un programa que demuestra cómo usar los valores de los índices. El usuario hace clic en un elemento del cuadro de lista, y el programa muestra el valor del índice equivalente en un cuadro de texto (Figura 13.4).

Cuando ocurre el evento del clic invocamos el siguiente método para manejarlo:

```
private void compras_SelectedIndexChanged(object sender, EventArgs e)
{
    textBox1.Text = Convert.ToString(compras.SelectedIndex);
}
```

`SelectedIndex` es una propiedad del cuadro de lista que proporciona el valor del índice del elemento en el que se hizo clic (o `-1` si no se ha hecho selección alguna). Al ejecutar este programa podemos ver que los valores de los índices no se almacenan como parte del cuadro de lista, sino que la computadora conoce los valores y éstos pueden usarse como y cuando sea necesario. También podemos confirmar que los valores de los índices empiezan en cero y no en uno.

## PRÁCTICA DE AUTOEVALUACIÓN

13.1 ¿Cuál es el valor del índice para el elemento pan en la Figura 13.4?

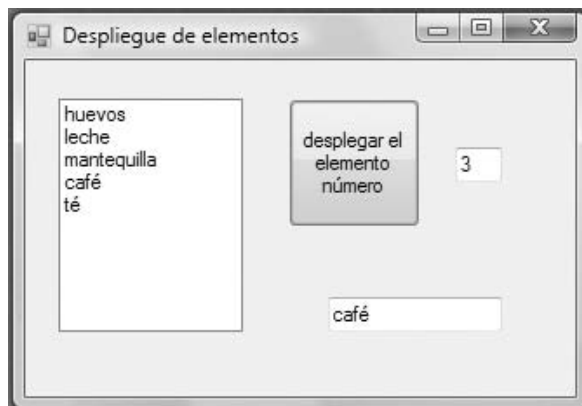


Figura 13.5 Despliegue de un elemento del cuadro de lista.

La Figura 13.5 muestra un programa que permite al usuario mostrar el valor en cierto índice específico. El código para manejar el clic del botón es:

```
private void button1_Click(object sender, EventArgs e)
{
    int índice;

    índice = Convert.ToInt32(índiceTextBox.Text);
    valueTextBox.Text = Convert.ToString(compras.Items[índice]);
}
```

Este programa obtiene el valor de un índice mediante un cuadro de texto, y convierte su representación `string` en un valor `int` a través del método `Convert.ToInt32`, para después colocarlo en la variable `índice`. A continuación se utiliza el valor del índice para acceder al elemento correspondiente en la lista. La notación utilizada sirve para proporcionar el nombre de la lista seguido del índice encerrado entre corchetes:

```
compras.Items[índice]
```

Por último, debemos convertir el valor del elemento del cuadro de lista en una cadena de texto mediante el método `ToString` antes de poder mostrarlo en un cuadro de texto. Esto se debe a que un objeto `List` es capaz de almacenar objetos de cualquier tipo concebible. Por ende, al remover un elemento de una lista debemos convertirlo nuevamente en el tipo deseado.

#### PRÁCTICA DE AUTOEVALUACIÓN

**13.2** ¿Qué elemento se encuentra en el índice con el valor 1 en la Figura 13.5?

## ● Eliminación de elementos de una lista

Ya hemos visto cómo agregar elementos a los cuadros de lista; veamos ahora de qué manera podemos eliminar información de ellos. El método `RemoveAt` de la clase `List` elimina el elemento que se encuentra en un valor de índice específico. Por lo tanto, si tenemos un cuadro de lista llamado `compras`, para eliminar el elemento en el índice con el valor 3, por ejemplo, emplearíamos la siguiente instrucción:

```
compras.Items.RemoveAt(3);
```

Al ejecutar esta instrucción también desaparece el vacío que queda al borrar el elemento de la lista.

## ● Inserción de elementos en una lista

Hemos comentado cómo agregar elementos al final de una lista mediante el método `Add`, pero también es posible insertarlos dentro del cuerpo del listado mediante el método `Insert`. Dada una lista existente, podemos hacer lo siguiente:

```
compras.Items.Insert(5, "té");
```

El elemento que se encontraba antes en el índice con el valor 5 se desplaza hacia abajo en la lista, junto con los elementos subsiguientes.

## ● Búsquedas rápidas

Los cuadros de lista son tablas susceptibles de utilizarse convenientemente para realizar búsquedas rápidas. Por ejemplo, podemos construir un cuadro de lista (Figura 13.6) que contenga los nombres de los meses, de enero a diciembre. Después, si alguien nos proporciona un mes expresado como número (del 1 al 12), empleamos la tabla para convertir el número en el texto equivalente.

La Figura 13.7 muestra la apariencia que tiene el programa para el usuario: el cuadro de lista es invisible (hemos establecido su propiedad `visible` en `false`), ya que no hay necesidad de que el usuario lo vea.

0	Enero
1	Febrero
2	Marzo
3	etc.

Figura 13.6 Diagrama de un cuadro de lista para convertir enteros en nombres de meses.

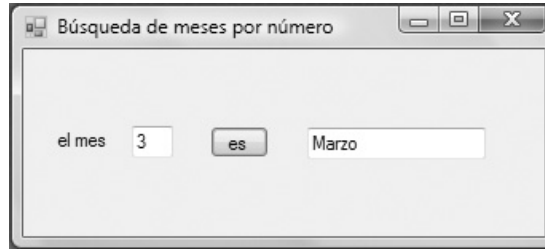


Figura 13.7 El programa Búsqueda de meses por número.

Al diseñar el programa introdujimos los valores Enero, Febrero, Marzo, etc., directamente en la propiedad `Items` del cuadro de lista.

Cuando lo ejecutamos, el programa convierte de la siguiente manera el número introducido mediante un cuadro de texto:

```
private void button1_Click(object sender, EventArgs e)
{
    int númeroMes;
    string nombreMes;

    númeroMes = Convert.ToInt32(númeroMesTextBox.Text);
    nombreMes = Convert.ToString(meses.Items[númeroMes - 1]);
    nombreMesTextBox.Text = nombreMes;
}
```

Los números que representan los meses van del 1 al 12, mientras que los valores de los índices empiezan desde 0. Por lo tanto, debemos restar 1 al número del mes, como se muestra en el código anterior, para convertirlo en un índice apropiado. La propiedad `Items` del cuadro de lista permite que el programa acceda al valor de su elemento llamado `meses`.

Utilizar una tabla de búsqueda rápida como la anterior es una alternativa con la que evitamos escribir doce instrucciones `if` o utilizar una instrucción `switch` con igual número de opciones.

## ● Operaciones aritméticas en cuadros de lista

A continuación examinaremos un cuadro de lista llamado `números`, y realizaremos operaciones aritméticas con los números enteros que incluye. Los cuadros de lista siempre contienen cadenas de caracteres, pero hay un tipo de cadena que utiliza dígitos, dando por resultado un número. La Figura 13.8 muestra un programa que permite al usuario introducir números en un cuadro de lista. Después un botón hace que aparezca la suma de los mismos, y otro provoca que se despliegue la cifra más alta de la lista.

El siguiente es el código que permite sumar todos los números en una lista. Como vamos a procesar todos los números del cuadro de lista podemos utilizar una instrucción `foreach` para ejecutar el ciclo. Este procedimiento es particularmente útil y conciso en una situación como ésta, pues extrae de manera automática cada elemento de la lista, uno a la vez. El siguiente paso de nuestro programa consiste en convertir cada valor de cadena de texto de la lista en un número entero, y sumarlo a un total acumulado, llamado `suma`, el cual en un principio es igual a 0. Por último se coloca el valor en un cuadro de texto.



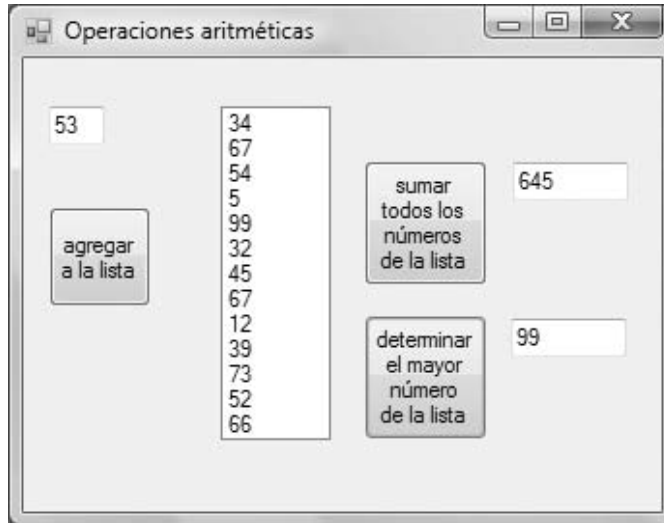


Figura 13.8 Operaciones aritméticas en un cuadro de lista.

```
private void botónSumar_Click(object sender, EventArgs e)
{
    int suma = 0;

    foreach ( string s in números.Items )
    {
        int número = Convert.ToInt32(s);
        suma = suma + número;
    }
    sumaTextBox.Text = Convert.ToString(suma);
}
```

Para continuar analizaremos un método cuyo propósito es encontrar el elemento más grande en una lista de números. Esto se realiza utilizando una variable llamada `mayor`, que se encargará de llevar un registro de los valores para encontrar el más alto. Al principio `mayor` es igual al valor en el índice 0 del cuadro de lista (el primer elemento de la lista). Este valor se copia de la lista, pero antes de poder usarlo debemos convertirlo en `string` mediante el método `Convert.ToString`, y luego en un `int` mediante el método `Convert.ToInt32`.

Emplearemos una instrucción `foreach` para procesar los números de la lista. Cada elemento de la misma se compara con el `mayor`, y si es más alto, el valor de `mayor` se actualizará. En consecuencia, cuando lleguemos al final de la lista, `mayor` contendrá el valor más grande.

```
private void botónMayor_Click(object sender, EventArgs e)
{
    int mayor;

    mayor = Convert.ToInt32(Convert.ToString(números.Items[0]));
    foreach ( string s in números.Items )
```

```
{  
    int número = Convert.ToInt32(s);  
    if (número > mayor)  
    {  
        mayor = número;  
    }  
}  
mayorTextBox.Text = Convert.ToString(mayor);  
}
```

### PRÁCTICA DE AUTOEVALUACIÓN

**13.3** Modifique este método, de manera que encuentre el valor más pequeño de la lista. Todo lo que necesita es una simple modificación.

Las dos secciones de que consta el código anterior ilustran una característica común de los programas que manipulan listas: cada vez que debamos procesar todos los elementos que conforman una lista, probablemente sea más apropiado utilizar una instrucción **foreach** en vez de una instrucción **for** o **while**.

## ● Búsquedas detalladas

Nuestro siguiente programa lleva a cabo búsquedas detalladas. Supongamos que contamos con una lista (por ejemplo, de compras) y queremos buscar cierto elemento en ella, para lo cual escribimos el nombre que nos interesa (por ejemplo, *café*) en un cuadro de texto, como se muestra en la Figura 13.9.

El programa examina todos los elementos de la lista desde el primero hasta el último, tratando de encontrar el que ha sido solicitado. Si no lo encuentra, el valor del índice se vuelve igual al tamaño de la lista (*longitud*), y el ciclo termina; si encuentra el elemento, la variable **bool** llamada **encontrado** adquiere el valor **true**, el ciclo se da por finalizado y se despliega un cuadro de mensaje informando al usuario que el elemento fue localizado.

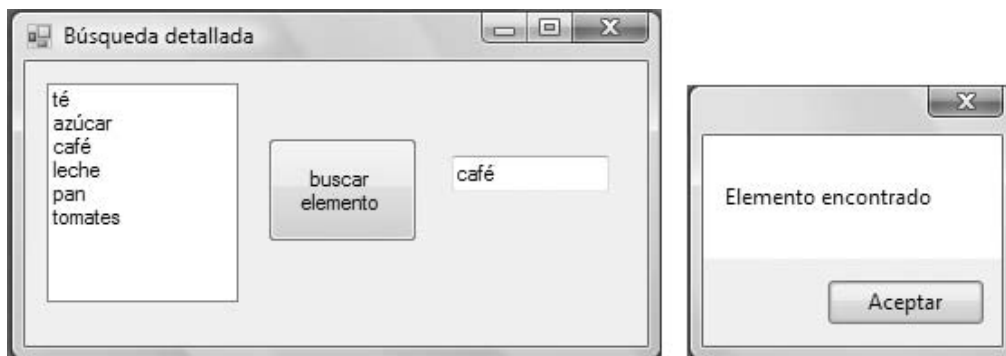


Figura 13.9 Búsqueda detallada en un cuadro de lista.

```

private void button1_Click(object sender, EventArgs e)
{
    int longitud;
    int índice;
    bool encontrado;
    string elementoDeseado;

    longitud = compras.Items.Count;
    elementoDeseado = textBox1.Text;
    encontrado = false;
    índice = 0;
    while ((encontrado == false) && (índice < longitud))
    {
        if (Convert.ToString(compras.Items[índice]) == elementoDeseado)
        {
            encontrado = true;
            MessageBox.Show("Elemento encontrado");
        }
        else
        {
            índice++;
        }
    }
}

```

Esta es una técnica de búsqueda serial clásica.

## ● Listas genéricas

Todos los ejemplos que hemos visto a lo largo del capítulo utilizan cuadros de lista, pero en realidad éstos emplean listas para guardar las cadenas de texto. En vista de lo anterior, enseguida comentaremos cómo emplear una lista directamente.

Suponga que necesitamos crear una lista llamada `miLista` para guardar varias cadenas de texto. Para ello utilizamos la siguiente instrucción:

```
List<string> miLista = new List<string>();
```

Aquí podemos ver que el nombre de la clase de los objetos que guardará la lista se coloca entre los signos < y > (en dos lugares). Esta característica se conoce como *genéricos*, y significa que podemos crear una lista personalizada para contener objetos de cierta clase específica; en este caso, cadenas de texto.

Una vez que hayamos creado una lista para guardar cadenas, podemos agregar algunos elementos:

```

private void botónAgregar_Click(object sender, EventArgs e)
{
    miLista.Add("pan");
    miLista.Add("leche");
    miLista.Add("café");
}

```

Si necesitamos mostrar el contenido de la lista tendremos que escribir código para hacerlo de manera explícita. Podríamos desplegarlo en un cuadro de lista, como se muestra a continuación, copiando los valores de las cadenas de texto uno a uno:

```
private void botónMostrar_Click(object sender, EventArgs e)
{
    foreach ( string s in miLista )
    {
        miListBox.Items.Add(s);
    }
}
```

Uno de los beneficios de las listas genéricas radica en que podemos crearlas para guardar objetos de cualquier clase deseada: cadenas de texto, botones o cualquier otra descrita por el programador. En el capítulo 24 veremos más detalles sobre el tema.

### Fundamentos de programación

- Los cuadros de lista constituyen probablemente el tipo de estructura de datos más simple que proporciona C#. Permiten ensamblar, visualizar y manipular listas de cadenas de texto. Una estructura de datos es un grupo de elementos de datos que se pueden procesar de manera uniforme. Como son visibles, los cuadros de lista representan un buen mecanismo para aprender sobre las estructuras de datos. Los cuadros de lista emplean un objeto lista para guardar la información. Este tipo de estructuras de datos se establecen en la memoria principal de la computadora (no en el almacenamiento secundario), de manera que sólo existen mientras se ejecuta el programa. Cuando éste termina, la estructura de datos se destruye.
- Los arreglos (tema que abordaremos en el capítulo 14) constituyen otro importante tipo de estructura de datos. Un arreglo es una colección de elementos de datos similares, cada uno de los cuales tiene su propio índice. A diferencia de los cuadros de lista, los arreglos son invisibles, de manera que para desplegar sus elementos el programador debe escribir instrucciones explícitas. Además, a diferencia de los cuadros de lista, los arreglos no permiten la inserción ni la eliminación de elementos.
- En circunstancias en las que debemos procesar todos los elementos de un cuadro de lista, la estructura de control natural que podemos utilizar es el ciclo `foreach`.

### Errores comunes de programación

Un error común consiste en pensar que los valores de los índices empiezan en 1 (en realidad comienzan en 0).

### Nuevos elementos del lenguaje

- La instrucción `foreach` para hacer repeticiones o iteraciones a través de todos los elementos en una lista.
- La notación `[]` para especificar un elemento de una lista.
- La notación `<>` para describir una lista genérica.

## Resumen

- Un cuadro de lista es un cuadro de GUI que contiene una lista de cadenas de texto.
- Cada uno de los elementos que conforman un cuadro de lista se identifica de manera única mediante un entero llamado *índice*. Los valores de los índices no se guardan, y siempre empiezan en 0.
- A través de la codificación apropiada podemos eliminar y modificar elementos de los cuadros de lista, además de agregarlos al final de la lista o insertarlos en el punto que deseemos.
- Los cuadros de lista emplean un objeto `List` para almacenar la información que contienen. Es el objeto `List` el que soporta los métodos para agregar y eliminar elementos en un cuadro de lista.

## EJERCICIOS

- 13.1** Escriba un programa en el que se elimine de inmediato un elemento de cuadro de lista seleccionado al hacer clic en él. Como alternativa proporcione un botón “eliminar” que permita borrar el elemento seleccionado.
- 13.2** Modifique el programa para que los elementos del cuadro de lista siempre se ordenen alfabéticamente de manera automática.
- 13.3** Agregue un botón que “vacíe” el cuadro de lista mediante el método `clear`.
- 13.4** Modifique el programa de manera que un elemento del cuadro de lista pueda sustituirse por algún otro texto. Por ejemplo, que “leche” se sustituya por “azúcar”. Proporcione un botón llamado “sustituir” que realice esta acción. El nuevo texto debe introducirse en un cuadro de texto.
- 13.5** Escriba un programa que permita insertar o eliminar elementos de cualquier posición en un cuadro de lista, mediante el uso de botones apropiados.
- 13.6** Mejore el método de búsqueda, de manera que aparezca un mensaje indicando si el elemento requerido se encontró o no en el cuadro de lista.

## SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

**13.1** 0

**13.2** leche

**13.3** Cambie el signo mayor que por el signo menor que.

# 14

## Arreglos

**En este capítulo conoceremos cómo:**

- declarar un arreglo;
- utilizar un índice;
- obtener el tamaño de un arreglo;
- pasar arreglos como parámetros;
- inicializar un arreglo;
- llevar a cabo operaciones típicas, como búsquedas rápidas y detalladas;
- crear arreglos de objetos.

### ● Introducción

---

Con la excepción de los cuadros de lista, a lo largo de este libro hemos venido analizando elementos de datos (variables) individuales y aislados. Por ejemplo:

```
int conteo, suma;  
string nombre;
```

Estos elementos de datos tienen “existencia” propia al desempeñar funciones útiles en los programas, como conteos, sumas, etcétera. Podemos considerar estas variables como lugares en la memoria que tienen nombres individuales.

En contraste, en la vida real es muy frecuente que tengamos que lidiar con datos que no están aislados, sino agrupados en colecciones de información. Algunas veces este tipo de información se encuentra almacenada en tablas, tal como ocurre en el caso de los itinerarios de trenes, los directorios telefónicos o los estados de cuenta bancarios. En programación a estas colecciones de datos se les conoce como estructuras de datos, y tienen la particularidad de que cada parte de la información de la tabla está relacionada de cierta forma con el resto. El arreglo es uno de los tipos más

simples de estructura de datos en programación. Un arreglo se puede considerar simplemente como una tabla con una sola fila (o columna) de información en forma de números, cadenas de caracteres o cualquier otra cosa.

En este capítulo veremos arreglos de números, de cadenas de caracteres y de otros objetos, como objetos gráficos.

He aquí un arreglo de números:

23	54	96	13	7	32
----	----	----	----	---	----

el cual podría representar las edades de un grupo de personas que participan en una fiesta. Por otro lado, la siguiente es una tabla de palabras que almacena los nombres de los miembros de un grupo musical:

John	Paul	George	Ringo
------	------	--------	-------

C# denomina estas tablas como arreglos. En programación cada uno de los datos de un arreglo es conocido como *elemento*, y nos referimos a él por la posición que ocupa en el arreglo llamada *índice*, (aunque en el mundo de la programación algunas veces se utiliza el término *componente* en vez de elemento, y el término *subíndice* en vez de índice). Desde nuestro punto de vista, el nombre John está en la posición uno de la tabla anterior, pero C# cuenta los elementos a partir de cero. En consecuencia, la cadena de caracteres Ringo se encuentra en la tercera posición del arreglo. A la posición que ocupa un elemento en un arreglo se le conoce como índice. Así, podemos visualizar los arreglos y sus índices de la siguiente manera:

arreglo:	John	Paul	George	Ringo
índices:	0	1	2	3

Recuerde que los índices no se guardan en la memoria de la computadora, sólo los datos. Los índices constituyen el mecanismo que nos permite localizar la información en un arreglo.

He aquí otro arreglo, esta vez conteniendo números. También se muestran sus índices:

arreglo:	34	78	54	12	21	35
índices:	0	1	2	3	4	5

En los programas (al igual que en la vida real) por lo general tenemos que realizar las siguientes operaciones con arreglos:

- Crear el arreglo: determinar cuál es su longitud y qué tipo de elementos almacenará.
- Colocar valores en el arreglo (equivalente, por ejemplo, a introducir números en un directorio telefónico personal).
- Desplegar en pantalla el contenido del arreglo (los arreglos se guardan en la memoria de la computadora y, por ende, son invisibles).
- Buscar en el arreglo cierto valor (algo similar a buscar en el itinerario de trenes una corrida en horario conveniente).
- Sumar el contenido del arreglo (equivalente a averiguar cuánto gastó un cliente en el supermercado).

En este capítulo veremos cómo llevar a cabo cada una de estas acciones por separado, para luego reunir las en un programa completo. Empezaremos por analizar los arreglos de números. Nuestro plan es desarrollar, a lo largo de este capítulo, un programa con la interfaz que se muestra en la Figura 14.1. En él, un arreglo guarda los datos de precipitación pluvial durante los siete días de la semana (de lunes a domingo). El usuario del programa puede modificar el valor de cualquier elemento del arreglo. Además, debe mostrarse el número mayor que se encuentre en el arreglo.

## ● Creación de un arreglo

En C# los arreglos se declaran de la misma manera que cualquier otro objeto: mediante la palabra clave `new`, y por lo general en la parte superior de una clase o de un método. Además, el programador debe darle nombre, como se muestra a continuación:

```
int[] edades = new int[6];  
string[] grupo = new string[4];
```

La variable `edades` está ahora lista para guardar un arreglo de enteros. Al igual que en el caso de cualquier otra variable, es común (y muy recomendable) que el nombre elegido para ella describa su función con claridad. Ese nombre identificará el arreglo completo, incluyendo toda la colección de datos que la conforman. Las reglas para elegir el nombre de un arreglo son iguales que para cualquier otra variable en C#. El número entre corchetes que está después del nombre del arreglo representa su tamaño.

El arreglo `edades` es lo bastante grande como para contener seis números (enteros, `int`), cuyos índices irán de 0 a 5.

Por su parte, el arreglo `grupo` tiene capacidad suficiente para contener cuatro cadenas de caracteres (`string`); sus índices van de 0 a 3.

### PRÁCTICA DE AUTOEVALUACIÓN

**14.1** Declare un arreglo para guardar los datos de precipitación pluvial para cada uno de los siete días de la semana.

## ● Índices

Para hacer referencia a un elemento individual de un arreglo el programa debe especificar el valor del índice (algunas veces llamado subíndice) correspondiente. Por lo tanto, `edades[3]` hace referencia al elemento que tiene el índice 3 en el arreglo: el valor 12 en el caso de nuestro ejemplo. De manera similar, `grupo[2]` contiene la cadena `George`. Recuerde que los índices empiezan en 0, por lo que un arreglo de longitud 4 tiene índices que van de 0 a 3. Por lo tanto, si hiciéramos referencia a `grupo[4]` estaríamos cometiendo un error, el programa se detendría y aparecería un mensaje indicándolo.



En resumen, los valores de los índices:

- empiezan en cero;
- son enteros;
- llegan hasta un número menos que el tamaño del arreglo (el valor que se especifica al momento de declarar el arreglo).

Algunas veces (como veremos más adelante) es conveniente utilizar como índice el valor de una variable. En tales casos utilizamos variables `int` como índices.

Podemos recibir el valor como entrada mediante un cuadro de texto, de esta manera:

```
edades[2] = Convert.ToInt32(textBox1.Text);
grupo[3] = textBox2.Text;
```

Y para desplegar los valores en pantalla procedemos de forma similar:

```
textBox3.Text = "la primera edad es " + Convert.ToString(edades[0]);
textBox4.Text = "el 4o. miembro del grupo es " + grupo[3];
```

Este último ejemplo ilustra lo cuidadosos que debemos ser al trabajar con índices de arreglos.

También, es posible modificar los valores de elementos individuales de un arreglo a través de instrucciones de asignación, como en el siguiente ejemplo:

```
edades[3] = 99;
grupo[2] = "Mike";
```

Para hacer referencia a los elementos individuales de un arreglo, en todos estos fragmentos de programa hemos especificado el valor de un índice.

## PRÁCTICA DE AUTOEVALUACIÓN

### 14.2 Dada la declaración:

```
int[] tabla = new int[3];
```

¿cuál es la longitud del arreglo y cuál es el rango válido de sus índices?

Es frecuente que necesitemos hacer referencia al  $n$ -ésimo elemento en un arreglo, siendo  $n$  una variable. Es en esta situación que resulta más claro cuán útiles son los arreglos. Por ejemplo, suponga que deseamos sumar todos los números de un arreglo (de números), e imagine que éste consta de siete elementos que representan el número de computadoras vendidas en un almacén durante cada día de la semana:

```
int[] venta = new int[7];
```

Para insertar valores en el arreglo emplearemos instrucciones de asignación. Suponga que el lunes (día 0) se vendieron 13 computadoras:

```
venta[0] = 13;
```

y el resto de la semana se vendieron estas cantidades:

```
venta[1] = 8;
venta[2] = 22;
venta[3] = 17;
venta[4] = 24;
venta[5] = 15;
venta[6] = 23;
```

Ahora queremos obtener la venta total de la semana. Una manera (poco hábil) de obtener ese resultado consistiría en escribir lo siguiente:

```
suma = venta[0] + venta[1] + venta[2] + venta[3]
      + venta[4] + venta[5] + venta[6];
```

Lo cual es correcto, pero no aprovecha la regularidad de un arreglo. La alternativa es usar un ciclo `for`. Una variable —digamos, `númeroDía`— se emplea para almacenar el valor del índice que representa el día de la semana. Al principio el índice se hace igual a 0, y su valor se incrementa cada vez que se repite el ciclo:

```
int suma = 0;
for ( int númeroDía = 0; númeroDía <= 6; númeroDía++ )
{
    suma = suma + venta[númeroDía];
}
```

En cada iteración del ciclo se suma al total el siguiente valor del arreglo. En realidad este fragmento de código no es más corto que la solución anterior, pero sin duda lo sería si el arreglo tuviera mil elementos que sumar. Otra ventaja es que el código muestra explícitamente que está realizando una operación sistemática en un arreglo.

Los índices son el único lugar de la programación en el que se permite (algunas veces) el uso de nombres un poco crípticos. Sin embargo, en el fragmento de programa anterior queda claro el uso como índice de `númeroDía`, y el nombre está estrechamente relacionado con el problema a resolver.

## PRÁCTICA DE AUTOEVALUACIÓN

---

### 14.3 ¿Qué hace el siguiente fragmento de código?

```
int[] tabla = new int[10];
for ( int índice = 0; índice <= 10; índice++ )
{
    tabla[índice] = índice;
}
```

## ● Longitud de los arreglos

Un programa en ejecución siempre puede conocer la longitud de los arreglos que intervienen en él. Por ejemplo, si tenemos un arreglo declarado de la siguiente forma:

```
int[] tabla = new int[10];
```

podemos conocer su longitud si utilizamos la propiedad `Length`, como en el siguiente ejemplo:

```
int tamaño;  
tamaño = tabla.Length;
```

En este caso, `tamaño` tiene un valor de 10. Tal vez le parezca inútil querer saber la longitud de un arreglo; después de todo tenemos que proporcionar ese dato al momento de declararlo. Sin embargo, más adelante comprobaremos la utilidad de esta herramienta.

Una vez que creamos un arreglo su longitud es fija. Los arreglos no son elásticos; es decir, no se ampliarán según sea necesario para almacenar información. Ésa es la razón por la que resulta imprescindible crear arreglos suficientemente grandes para satisfacer las necesidades de un programa específico. Lo que sí es posible es utilizar la misma variable si creamos un nuevo arreglo de distinto tamaño mediante el uso de `new`; aunque, al hacerlo perderemos los datos del arreglo original.

Al diseñar un nuevo programa debemos tener en cuenta el tamaño de cualquier arreglo que intervenga en él. En ocasiones la naturaleza del problema hace que esto resulte obvio. Por ejemplo, si los datos se relacionan con los días de la semana, sabemos que el arreglo constará de siete elementos. Sin embargo, hay otros tipos de estructuras de datos (como una lista de arreglos) que se expanden y contraen pieza por pieza, según sea necesario.

## ● Paso de arreglos como parámetros

Como vimos en capítulos anteriores, los métodos son muy importantes en la programación. Un aspecto importante de su uso involucra la acción de pasarles información en forma de parámetros, y devolver un valor. Es por ello que a continuación explicaremos cómo pasar y devolver arreglos a los métodos.

Suponga que queremos escribir un método cuyo trabajo sea calcular la suma de los elementos que se encuentran en un arreglo de enteros. Siendo programadores perceptivos, deseamos que el método sea de propósito general para que pueda lidiar con arreglos de cualquier longitud. Esto es sencillo, ya que el método puede encontrar fácilmente la longitud del arreglo. Así, el parámetro que debe pasarse al método es simplemente el arreglo, y el resultado que se devolverá al usuario del método es un número: la suma de los valores.

El siguiente es un ejemplo de cómo invocar este método:

```
int[] tabla = new int[24];  
int total;  
  
total = Suma(tabla);
```

El método en sí sería:

```
private int Suma(int[] arreglo)
{
    int total = 0;
    for ( int índice = 0; índice < arreglo.Length; índice++)
    {
        total = total + arreglo[índice];
    }
    return total;
}
```

Observe que en el encabezado del método el parámetro se declara como un arreglo, con corchetes, pero no hay parámetro que indique la longitud que éste debe tener. Para averiguar su tamaño, el método utiliza la propiedad `Length`. Como puede aceptar un arreglo de cualquier longitud, este método es de propósito general y puede ser muy útil. Esto es mucho más conveniente que un método de propósito específico, que sólo funcione cuando el arreglo conste, digamos, de ocho elementos.

#### PRÁCTICA DE AUTOEVALUACIÓN

**14.4** Escriba un método que muestre un arreglo de enteros, uno por línea, en un cuadro de texto multilínea. El único parámetro del método debe ser el arreglo.

### ● Uso de constantes

Un programa con varios arreglos debe incluir las declaraciones de éstos, y probablemente también varios ciclos `for`. Los arreglos, junto con sus longitudes, se pasan al programa como parámetros. Bajo tales condiciones hay muchas probabilidades de confusión, en especial si dos arreglos distintos tienen la misma longitud.

Suponga, por ejemplo, que vamos a escribir un programa para analizar las calificaciones obtenidas por un grupo de estudiantes en sus tareas. Imagine que hay diez estudiantes y necesitamos un arreglo para guardar la calificación promedio para cada uno de ellos:

```
int[] califEstudiante = new int[10];
```

Da la casualidad de que también son diez las materias que cursan, de manera que necesitamos un segundo arreglo para almacenar la calificación promedio en cada materia:

```
int[] califMateria = new int[10];
```

El problema es que cada vez que vemos el número 10 en el programa, no sabemos si se refiere a la cantidad de estudiantes o de materias. En este caso en particular no importa, ya que son iguales. Pero suponga que necesitamos modificar el programa de manera que trabaje con veinte estudiantes. Sería muy conveniente poder cambiar cada ocurrencia del número 10 por el número 20, utili-

zando la función de reemplazar en el entorno de desarrollo integrado. Pero como los arreglos tienen la misma longitud, esto podría causar daños fuertes al programa.

Una manera de aclarar esta situación consistiría en declarar las longitudes de los arreglos como constantes, y después utilizar esas constantes en ciclos `for`, por ejemplo:

```
const int estudiantes = 20;
const int materias = 24;
```

Y luego podemos emplear las constantes de esta manera:

```
int[] califEstudiante = new int[estudiantes];
int[] califMateria = new int[materias];

for ( int índice = 0; índice < estudiantes; índice++ )
{
    // cuerpo del ciclo
}
```

Con esta solución podríamos realizar cambios en el programa sin ocasionar problemas, modificando un solo número en la declaración de las constantes.

#### PRÁCTICA DE AUTOEVALUACIÓN

**14.5** Escriba el código para colocar ceros en cada elemento del arreglo `califMateria`.

## ● Inicialización de arreglos

Como sabemos, inicializar significa dar un valor inicial a una variable. Por ejemplo, si usted escribe lo siguiente:

```
int[] tabla = new int[10];
```

se crea en la memoria un arreglo con sus elementos inicializados con ceros. Esto se debe a que cuando el programador no proporciona explícitamente valores iniciales, el compilador inserta valores predeterminados: ceros para los números, "" para las cadenas de caracteres, y `null` para los objetos.

Una forma común de inicializar un arreglo de manera explícita es al momento de declararlo. Los valores iniciales requeridos se encierran entre llaves y se separan mediante comas. Pero el tamaño del arreglo *no* debe proporcionarse en su posición usual. La siguiente inicialización:

```
int[] edades = {23, 54, 96, 13, 7, 32};
```

en donde la longitud del arreglo no se da de manera explícita, es equivalente a:

```
int[] edades = new int[6];
edades[0] = 23;
edades[1] = 54;
edades[2] = 96;
edades[3] = 13;
edades[4] = 7;
edades[5] = 32;
```

He aquí otro ejemplo; en este caso se inicializa un arreglo de cadenas de caracteres:

```
string[] grupo = {"John", "Paul", "George", "Ringo"};
```

Otra forma de inicializar un arreglo es mediante un ciclo, como en el siguiente ejemplo:

```
int[] tabla = new int[25];  
for ( int índice = 0; índice < tabla.Length; índice++ )  
{  
    tabla[índice] = 0;  
}
```

Si el programa necesita restablecer periódicamente el arreglo a sus valores iniciales, podemos utilizar el ciclo `for` anterior.

#### PRÁCTICA DE AUTOEVALUACIÓN

**14.6** Declare un arreglo llamado `números` con longitud de cinco enteros, y llénela con los números del 1 al 5 como parte de la declaración.

### ● Un programa de ejemplo

A continuación combinaremos todos los conceptos que hemos explicado, para construir un programa que recibe varios números, los coloca en un arreglo y luego los despliega en pantalla. En la Figura 14.1 aparece la interfaz de este programa. Los datos desplegados representan la precipitación pluvial durante los siete días en una semana (de lunes a domingo). El usuario del programa

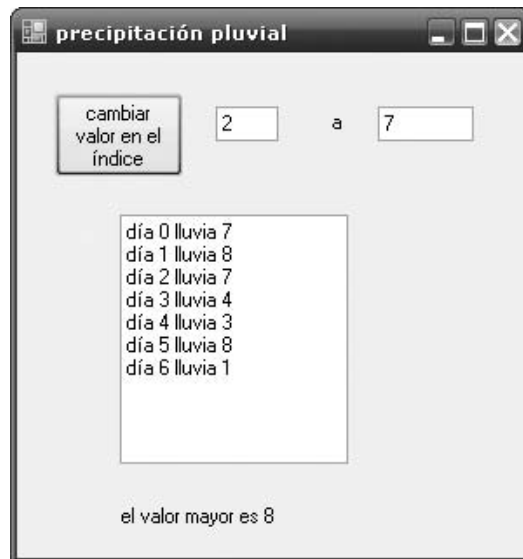


Figura 14.1 La interfaz del programa de precipitación pluvial.

introduce el número de un día en un cuadro de texto, y un valor de precipitación pluvial en otro. En respuesta, se despliega el mayor de los valores de precipitación pluvial.

Para codificar el programa primero se declara el arreglo en la parte superior de la clase del formulario. Sus valores se inicializan con una selección de valores.

```
private int[] lluvia = {7, 8, 0, 4, 3, 8, 1};
```

Luego declaramos el código para mostrar los valores del arreglo en un cuadro de texto multilínea:

```
private void Mostrar()
{
    lluviaTextBox.Clear();
    for ( int númeroDía = 0; númeroDía <= 6; númeroDía++ )
    {
        lluviaTextBox.AppendText("día " + Convert.ToString(númeroDía)
                                + " lluvia " + Convert.ToString(lluvia[númeroDía])
                                + "\r\n");
    }
}
```

El siguiente es el código para colocar un nuevo valor en un elemento del arreglo. El valor del índice está en un cuadro de texto; el valor de los datos está en otro. Al final se invoca el método **Mostrar** para desplegar el valor actualizado, y después al método **Mayor** para mostrar el valor más grande.

```
private void NuevoValor()
{
    int índice;
    int datos;
    índice = Convert.ToInt32(índiceTextBox.Text);
    datos = Convert.ToInt32(valorTextBox.Text);
    lluvia[índice] = datos;
    Mostrar();
    Mayor();
}
```

En este caso el método **NuevoValor** se debe invocar al hacer clic en el botón del formulario. Continuemos con el código para calcular el valor de precipitación pluvial más alto entre los datos del arreglo. El método que utilizamos asume, para empezar, que el primer elemento es el mayor. Después analizamos el resto de los elementos por turno, comparándolos con ese valor mayor. Si encontramos un valor más alto que el que ya tenemos, el valor mayor se actualiza. Éste es un método clásico.

```
private void Mayor()
{
    int mayor;

    mayor = lluvia[0];
    for ( int índice = 0; índice <= 6; índice++ )
    {
        if (mayor < lluvia[índice])
```

```

        {
            mayor = lluvia[índice];
        }
    }
    etiquetaMayor.Text = "el valor mayor es " + Convert.ToString(mayor);
}

```

Cabe mencionar que es muy común utilizar la instrucción `for` junto con los arreglos; “son uno mismo”, como dice la canción. Desde luego, esto se debe a que un ciclo `for` aprovecha al máximo la uniformidad de los arreglos.

#### PRÁCTICA DE AUTOEVALUACIÓN

**14.7** Escriba un método para calcular y mostrar la precipitación pluvial total de la semana.

### ● Búsqueda rápida (accesos directos)

Parte de la utilidad de los arreglos radica en que nos permiten hacer accesos directos y fáciles entre sus elementos. Por ejemplo, en nuestro programa de práctica podemos extraer el valor de la precipitación pluvial correspondiente al martes, con sólo hacer referencia a `lluvia[1]`. Lo mismo se aplica a cualquier información a la que se pueda hacer referencia mediante un índice entero. He aquí un ejemplo más: si tenemos una tabla que muestre la altura promedio de un grupo de personas de acuerdo con su edad, podemos indizar la tabla usando una edad (digamos, 25):

```

double[] altura = new double[100];

double miAltura;
miAltura = altura[25];

```

De manera similar, si hemos enumerado los días de la semana del 0 al 6, podemos convertir un número en una cadena de caracteres mediante el siguiente código:

```

int númeroDía;
string nombreDía;
string[] nombre = {"Lunes", "Martes", "Miércoles", "Jueves",
                  "Viernes", "Sábado", "Domingo"};

nombreDía = nombre[númeroDía];

```

También podríamos lograr esto mediante una instrucción `switch`, pero el código sería más extenso y probablemente más complicado.

El uso de arreglos para realizar accesos directos es extremadamente útil y simple, y aprovecha al máximo su poder.

#### PRÁCTICA DE AUTOEVALUACIÓN

**14.8** Vuelva a escribir la conversión anterior, utilizando ahora una instrucción `switch`.



## ● Búsqueda detallada

Otra manera de acceder a la información que se encuentra en un arreglo es buscándola. Eso es lo que hacemos al consultar un directorio telefónico o un diccionario. Ya que, es imposible utilizar un índice de forma directa, porque no se conoce su posición. El ejemplo que consideraremos a continuación es un directorio telefónico (Figura 14.2).

Para comenzar estableceremos dos arreglos, uno para guardar nombres y el otro para almacenar los números telefónicos correspondientes:

```
private string[] nombres = new string[20];  
private string[] números = new string[20];
```

Una vez creados los arreglos, podemos colocar datos en ellos:

```
nombres[0] = "Alejandro";  
números[0] = "2720774";  
  
nombres[1] = "Marcela";  
números[1] = "5678554";  
  
nombres[2] = "FIN";
```

Ahora bien, una forma simple y efectiva de buscar en el directorio es empezar desde el principio y recorrer un elemento tras otro hasta llegar al nombre que estamos buscando. Sin embargo, podría darse el caso de que éste no se halle en el directorio, y debemos tener en cuenta esa posibilidad. Así, nuestra búsqueda continuará hasta encontrar lo que buscamos o hasta llegar al final de las entradas. ¿Cómo saber cuando no hay más nombres en el arreglo? Podríamos hacer una comprobación, pero un método más conveniente sería colocar una entrada especial en el arreglo para indicar el final de los datos útiles. Como se muestra en el código anterior, este marcador final consistirá en una entrada con el nombre **FIN**. Ahora podemos escribir el ciclo para buscar el número telefónico que deseamos.



Figura 14.2 Un directorio telefónico.

```
private void Buscar()
{
    int índice;
    string buscado;

    buscado = textBox1.Text;
    índice = 0;
    for ( índice = 0;
        nombres[índice] != buscado && nombres[índice] != "FIN";
        índice++ )
    {
    }
    if (nombres[índice] == buscado)
    {
        label1.Text = "el número es " + Convert.ToString(números[índice]);
    }
    else
    {
        label1.Text = "no se encontró el nombre";
    }
}
```

A esto se le conoce como búsqueda *serial*: empieza al principio del arreglo, en el elemento cero, y continúa buscando elemento por elemento, agregando uno al índice. La búsqueda sigue hasta que se halla el elemento deseado o se llega al nombre especial **FIN**.

Este tipo de búsqueda no hace suposición alguna en cuanto al orden de los elementos de la tabla, así que éstos pueden estar en cualquier orden. Otras técnicas de búsqueda aprovechan el orden de los elementos, como el alfabético; sin embargo, no las abordaremos aquí, ya que están más allá del alcance de este libro.

Por lo general, la información (como los números telefónicos) se almacena en un archivo en vez de hacerlo en un arreglo, ya que de esta forma se garantiza su permanencia. En consecuencia, casi siempre la información requerida se busca en el archivo en vez de hacerlo en un arreglo. Otra alternativa es introducir el archivo en la memoria, guardando la información en un arreglo y realizar la búsqueda cómo se mostró antes.

## ● Arreglos de objetos

---

Los arreglos pueden guardar cualquier tipo de información: enteros, números de punto flotante, cadenas de caracteres, botones, barras de seguimiento, cualquier objeto de la biblioteca o construido por el programador. La única restricción es que todos los objetos guardados en un arreglo deben ser del mismo tipo. En el siguiente ejemplo crearemos un arreglo de objetos globo (Figura 14.3). En capítulos anteriores del libro describimos este tipo de objeto.

Un objeto globo (que en realidad es sólo un círculo) tiene un tamaño y una posición en la pantalla. Además, se proporcionan métodos como parte del comportamiento del objeto para moverlo, modificar su tamaño y desplegarlo en pantalla. He aquí la descripción de la clase, recuerde que para poder utilizar los elementos gráficos dentro de la clase, debe agregar la línea “`using System.Drawing;`” en la parte superior de la clase:

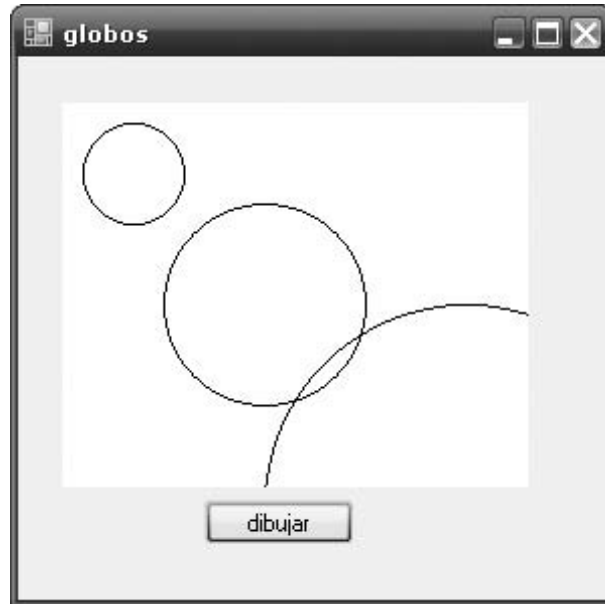


Figura 14.3 Trazo de un arreglo de globos.

```
public class Globo
{
    private int x;
    private int y;
    private int diámetro;

    public Globo(int xInicial, int yInicial, int diámetroInicial)
    {
        x = xInicial;
        y = yInicial;
        diámetro = diámetroInicial;
    }

    public void CambiarTamaño(int cambio)
    {
        diámetro = diámetro + cambio;
    }

    public void Mostrar(Graphics áreaDibujo, Pen lápiz)
    {
        áreaDibujo.DrawEllipse(lápiz, x, y, diámetro, diámetro);
    }
}
```

Ahora podemos crear un arreglo de globos:

```
private Globo[] fiesta = new Globo[10];
```

No obstante, esta instrucción sólo crea el arreglo, listo para guardar todos los globos. Nuestra siguiente tarea será crear algunos globos, como se muestra a continuación:

```
fiesta[0] = new Globo(10, 10, 50);
fiesta[1] = new Globo(50, 50, 100);
fiesta[2] = new Globo(100, 100, 200);
```

y desplegarlos:

```
private void MostrarGlobos()
{
    Graphics papel;
    papel = pictureBox1.CreateGraphics();
    Pen lápiz = new Pen(Color.Black);
    papel.Clear(Color.White);
    for ( int b = 0; b <= 2; b++ )
    {
        fiesta[b].Mostrar(papel, lápiz);
    }
}
```

La ventaja de almacenar los objetos globo en un arreglo radica en que eso nos permite llevar a cabo acciones con todos ellos como conjunto. Por ejemplo, podemos modificar el tamaño de todos los globos a la vez:

```
for ( int b = 0; b <= 2; b++ )
{
    fiesta[b].CambiarTamaño(20);
}
```

Para terminar, antes mencionamos que todos los elementos contenidos en el arreglo tienen que ser del mismo tipo, pero hay una excepción: si declara un arreglo de objetos de la clase `Object` podrá colocar distintos tipos de objetos en el arreglo. Esto se debe a que `Object` es la superclase de cualquier otra clase.

## Fundamentos de programación

- Los arreglos son colecciones de datos agrupados bajo un solo nombre. Todos los elementos de un arreglo son del mismo tipo. Los elementos individuales se identifican mediante un índice, un entero. Por ejemplo, si tenemos un arreglo llamado `tabla`, podemos hacer referencia a uno de sus elementos individuales como `tabla[2]`, en donde 2 es el índice. Además, podemos hacer referencia de manera similar a un elemento de un arreglo utilizando una variable entera como el índice; por ejemplo, `tabla[índice]`. Esta característica es la que hace a los arreglos tan útiles. Una vez creado, la longitud del arreglo permanece fija.

- Los arreglos pueden guardar datos de cualquier tipo; por ejemplo, `int`, `double`, `bool`, `Button`, `TextBox` (sin embargo, en cualquier arreglo todos los datos deben ser del mismo tipo).
- El arreglo es la estructura de datos más antigua y utilizada. Es compacta y permite acceder a ella con mucha rapidez mediante instrucciones de la computadora.
- Es común utilizar el ciclo `for` junto con los arreglos.

## Errores comunes de programación

Un error común en C# es confundir la longitud de un arreglo con su rango de índices válidos. Por ejemplo, el arreglo:

```
int[] tabla = new int[10];
```

tiene 10 elementos. El rango válido de índices para este arreglo es de 0 a 9. De acuerdo con lo anterior, `tabla[10]` sería una referencia a un elemento del arreglo que simplemente no existe. Por fortuna el sistema de C# comprueba violaciones de este tipo cuando el programa se ejecuta, y emite un mensaje de error en consecuencia.

He aquí un ejemplo común de cómo cometer errores al trabajar con arreglos:

```
int[] tabla = new int[10];

for ( int índice = 0; índice <= 10; índice++ ) // advertencia, hay un error
{
    tabla[índice] = 0;
}
```

El código anterior colocará un cero en todos los elementos del arreglo `tabla`, pero después trata de colocar un cero en la variable que se encuentre inmediatamente después del arreglo en la memoria de la computadora. En este caso el programa falla y despliega un mensaje `'IndexOutOfRangeException'` (excepción de índice fuera de rango). Siempre es conveniente comprobar cuidadosamente la condición para terminar un ciclo `for` cuando lo utilizamos con un arreglo.

Algunas veces los estudiantes tienen dificultades para visualizar en dónde está un arreglo. Ésta se guarda en la memoria principal; es invisible y sólo existe mientras el programa se ejecuta.

## Secretos de codificación

Un arreglo con 20 elementos se declara de la siguiente forma:

```
double[] tabla = new double[20];
```

Para hacer referencia a un elemento del arreglo, el índice se escribe entre corchetes, como en el siguiente ejemplo:

```
tabla[3] = 12.34;
```

## Resumen

- Un arreglo es una colección de datos a la que el programador le asigna un nombre. Todos los elementos que conforman un arreglo deben ser del mismo tipo (por ejemplo, todos `int`).
- Un arreglo se declara (al igual que otras variables) de la siguiente forma:

```
int[] pedro = new int[25];
```

en donde 24 es el valor del último índice. El arreglo consta de 25 elementos.

- Para hacer referencia a un elemento individual de un arreglo utilizamos un índice entero; por ejemplo:

```
pedro[12] = 45;
```

- Los índices tienen valores que empiezan en cero y llegan a su valor máximo, que es una unidad menos que su tamaño.

## EJERCICIOS

### Juegos

- 14.1 Nim** En este juego compite un usuario contra la computadora. Al inicio del juego hay tres pilas de cerillos o fósforos. En cada pila hay un número aleatorio de cerillos en el rango de 1 a 20. Las tres pilas se muestran en pantalla durante todo el juego. Una opción aleatoria determina quién va primero. Los jugadores toman turnos para extraer todos los cerillos que deseen de cualquier pila, pero sólo de una. Cada jugador debe extraer por lo menos un cerillo. El ganador es quien haga que su contrincante tome el último cerillo. Haga que la computadora juegue al azar; es decir, que seleccione una pila al azar y después una cantidad aleatoria de cerillos con base en los que haya disponibles.
- 14.2 Bóveda de seguridad** Establezca un arreglo que contenga los seis dígitos para abrir una bóveda de seguridad. Pida al usuario que introduzca seis números, uno a la vez, mediante botones etiquetados con los dígitos 0 al 9, y compruebe que sean correctos. Al introducir un dígito indique al usuario si es correcto o no; éste tendrá tres oportunidades antes de verse obligado a empezar desde cero otra vez.
- 14.3 Veintiuno** Escriba un programa para jugar con este juego de cartas, al que a veces también se le llama *blackjack*, *ving-et-un* o *pontoon*. La computadora debe actuar como el repartidor. Al principio éste entregará dos cartas al azar (en el juego real el repartidor tiene una enorme cantidad de cartas provenientes de varios mazos barajados). Su objetivo es obtener una puntuación más alta que la del repartidor, sin pasar de veintiún puntos. El as cuenta como uno o como once. En cualquier momento usted puede “pedir” (lo cual significa que desea otra carta), o “plantarse”, lo cual implica que está conforme con lo que tiene. También puede “pasarse”, es decir, rebasar los veintiún puntos. Si al final se planta o se pasa, será turno del repartidor para recibir sus propias cartas. El objetivo del repartidor es obtener una puntuación mayor a la de usted, sin pasarse; sin embargo, él ignora la puntuación que usted tiene, de manera que tendrá que arriesgarse y tratar de mejorar su juego.
- Proporcione botones para empezar un nuevo juego, pedir y plantarse. Al final de un juego los dos conjuntos de cartas que se repartieron deberán desplegarse.

## Operaciones básicas con arreglos

- 14.4 Datos pluviales** Complete el programa para manejar los datos de precipitación pluvial; incluya las siguientes operaciones:
- Suma de los valores y despliegue del total.
  - Búsqueda de los valores más grande y más pequeño, y despliegue de los mismos en pantalla.
  - Búsqueda del índice correspondiente al valor más grande.
- 14.5 Arreglo de cadenas** Escriba un programa que utilice un arreglo de 10 cadenas de caracteres. Escriba métodos que realicen cada una de las siguientes operaciones:
- Introducir valores con el teclado mediante un cuadro de texto.
  - Mostrar los valores en pantalla (para poder comprobar que se hayan introducido correctamente en el arreglo).
  - Introducir una palabra mediante un cuadro de texto, y buscarla para ver si está presente en el arreglo.
  - Desplegar un mensaje que diga si la palabra está en el arreglo o no.
- 14.6 Gráfica de barras** Las gráficas de barras son útiles para presentar datos como los de la precipitación pluvial, o los cambios en los precios de los bienes raíces. Escriba un método que muestre una gráfica de barras de los datos que reciba mediante un arreglo. El arreglo almacena varios valores, por ejemplo sobre la precipitación pluvial en cada uno de los siete días de la semana. Puede utilizar el método `FillRectangle` para trazar las barras individuales.
- 14.7 Gráfica circular** Las gráficas circulares (o de pastel) muestran cantidades a manera de proporciones y son, por lo tanto, muy útiles para presentar datos como presupuestos personales o empresariales. Escriba un método que despliegue en pantalla una gráfica circular de los datos que reciba mediante un arreglo. El arreglo debe contener las cantidades gastadas (por ejemplo) en viajes, alimentos, alojamiento, etc. Investigue el método `FillPie` de la biblioteca, pues le será muy útil en este programa.
- 14.8 Trazador de gráficas** Escriba un método para dibujar una gráfica a partir de los datos que se proporcionan mediante un arreglo de coordenadas *x*, y un arreglo de las coordenadas correspondientes *y*. El método debe tener el siguiente encabezado:

```
private void DibujarGráfica(double[] x, double[] y)
```

El propósito del programa es trazar líneas rectas de un punto a otro, y también debe dibujar los ejes.

## Estadísticas

- 14.9** Escriba un programa que introduzca una serie de enteros en un arreglo. Los números deben estar en el rango de 0 a 100. Calcule y despliegue en pantalla:
- el número más grande;
  - el número más pequeño;
  - la suma de los números;
  - la media de los números.

Muestre un histograma (gráfica de barras) que indique cuántos números se encuentran en los rangos 0 a 9, 10 a 19, etc.

### Números aleatorios

- 14.10** Verifique que la clase generadora de números aleatorios (capítulo 6) funcione correctamente. Configúrela para que proporcione números aleatorios en el rango 1 a 100. Después invoque el método cien veces y coloque las frecuencias en un arreglo, como en el ejercicio anterior. Por último, despliegue en pantalla el histograma de frecuencias, tal como hicimos en el ejercicio anterior. Dado que los números deben ser aleatorios, la altura de las barras del histograma será más o menos igual.

### Palabras

- 14.11 Organización de palabras** Escriba un programa que reciba como entrada cuatro palabras y después despliegue todas las permutaciones posibles. Por ejemplo, si se introducen las palabras malo, perro, muerde y hombre, en pantalla debería aparecer algo como esto:

```
hombre muerde perro malo
hombre malo muerde perro
perro muerde hombre malo
etc.
```

(¡No es necesario que todos los enunciados tengan sentido!)

### Procesamiento de información: búsqueda

- 14.12 Diccionario** Establezca dos arreglos que contengan pares de palabras cuyo significado sea equivalente en inglés y español. Después introduzca una palabra en inglés, busque su equivalente en español y despléguelo en pantalla. Asegúrese de comprobar que la palabra esté en el diccionario. Después agregue lo necesario para traducir en sentido opuesto, utilizando los mismos datos.

- 14.13 Biblioteca** Cada uno de los usuarios de una biblioteca tiene un código de identificación único, un entero. Cuando alguien desea pedir un libro prestado, se comprueba que su código de usuario sea válido.

Escriba un programa que busque la identificación de un usuario específico dentro de una tabla de códigos. El programa debe desplegar un mensaje que indique si el código es válido o no.

- 14.14 Directorio telefónico** Mejore el programa del directorio telefónico que vimos antes en este capítulo, de manera que puedan agregarse nuevos nombres y números. Después añada lo necesario para eliminar un nombre y un número.

### Procesamiento de información: ordenamiento

- 14.15 Ordenamiento** Escriba un programa que reciba como entrada una serie de números, los ordene de manera ascendente y los despliegue en pantalla.

Este programa no es fácil de escribir. Hay varias metodologías para ordenar datos; de hecho existen libros completos sobre el tema. Uno de los procedimientos más comunes es el siguiente.

Busque el número más pequeño del arreglo. Intercámbielo con el primer elemento. Ahora el primer elemento está en la posición correcta. Déjelo donde está y repita la operación con el



resto del arreglo (todos los elementos, excepto el primero). Repita la operación con un arreglo cada vez más pequeño hasta que esté completamente ordenado.

## Arreglos de objetos

**14.16 Globos** Amplíe el programa que almacena un arreglo de globos. Agregue la funcionalidad necesaria para:

- reventar todos los globos con base en un factor aleatorio;
- mover todos los globos la misma cantidad de pixeles.

**14.17 Directorio telefónico** Escriba un programa para crear y mantener un directorio telefónico. Cada elemento del arreglo debe ser un objeto de la clase `Entrada`:

```
public class Entrada
{
    private string elNombre;
    private string elNúmero;

    // propiedades para acceder al nombre y al número
}
```

Complete la clase `Entrada` y después cree el siguiente arreglo:

```
private Entrada[] directorio = new Entrada[1000];
```

Ahora coloque datos en él como se muestra a continuación, utilizando las propiedades:

```
directorio[0].Nombre = "Agustín Ramírez";
directorio[0].Número = "01 0114 255 3103";
```

Proporcione una GUI para introducir datos en el directorio. Provea una herramienta de búsqueda, de manera que si se escribe un nombre en un cuadro de texto, en otro aparezca el número telefónico correspondiente.

**14.18 Juego de cartas** Éste es un ejemplo que podría ser parte de un juego de cartas. Cada carta se describe mediante la siguiente clase:

```
public class Carta
{
    private int rango;
    private int palo;

    // propiedades para acceder al rango y al palo
}
```

Complete la clase `Carta`. Después cree un arreglo para almacenar un mazo completo de cartas:

```
private Carta[] mazo = new Carta[52];
```

Inicialice el mazo mediante un ciclo `for` para recorrer los cuatro palos y un ciclo `for` para recorrer los distintos rangos de las cartas.

**SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**

**14.1**    `int[] lluvia = new int[7];`

**14.2** El arreglo tiene tres elementos de longitud. Los índices válidos son de 0 a 2.

**14.3** El fragmento de programa coloca los números 0 a 10 en el arreglo, pero trata de acceder a un elemento inexistente con el valor de índice 10. Por lo tanto, el programa fallará.

**14.4**    `private void Mostrar(int[] arreglo)`  
      `{`  
          `textBox1.Clear();`  
          `for ( int índice = 0; índice < arreglo.Length; índice++ )`  
          `{`  
              `textBox1.AppendText(Convert.ToString(arreglo[índice])`  
                  `+ "\r\n");`  
          `}`  
      `}`

**14.5**    `for ( int índice = 0; índice < materias; índice++ )`  
      `{`  
          `califMateria[índice] = 0;`  
      `}`

**14.6**    `int[] números = {1, 2, 3, 4, 5};`

**14.7**    `private void TotalSemana()`  
      `{`  
          `int total = 0;`  
          `for ( int índice = 0; índice <= 6; índice++ )`  
          `{`  
              `total = total + lluvia[índice];`  
          `}`  
          `etiquetaTotal.Text = "el total es " + Convert.ToString(total);`  
      `}`

**14.8**    `switch (númeroDía)`  
      `{`  
          `case 0:`  
              `nombreDía = "Lunes";`  
              `break;`  
          `case 1:`  
              `nombreDía = "Martes";`  
              `break;`  
          `case 2:`  
              `nombreDía = "Miércoles";`  
              `break;`  
      `}`

```
case 3:
    nombreDía = "Jueves";
    break;
case 4:
    nombreDía = "Viernes";
    break;
case 5:
    nombreDía = "Sábado";
    break;
case 6:
    nombreDía = "Domingo";
    break;
}
```

# Arreglos bidimensionales (matrices)

En este capítulo conoceremos cómo:

- declarar un arreglo bidimensional;
- utilizar los índices de los arreglos bidimensionales;
- obtener la longitud de un arreglo bidimensional;
- pasar arreglos bidimensionales como parámetros;
- inicializar un arreglo bidimensional.

## ● Introducción

---

Los arreglos bidimensionales, también conocidos como tablas o **matrices** (por su parecido al concepto matemático), son muy comunes en la vida diaria; los siguientes ejemplos son algunas representaciones:

- un tablero de ajedrez;
- un itinerario de trenes;
- una hoja de cálculo.

En el capítulo anterior hablamos sobre los arreglos unidimensionales. C# ofrece una extensión natural de arreglos unidimensionales a bidimensionales. Por ejemplo, la siguiente declaración:

```
int[,] ventas = new int[4, 7];
```

declara un arreglo bidimensional, o matriz, de enteros. Siguiendo con nuestro ejemplo del capítulo previo, esta matriz contiene las cifras de ventas de computadoras en cuatro almacenes durante cada uno de los siete días de una semana (Figura 15.1). La matriz se llama **ventas**. En términos de tabla, podemos considerar que consta de cuatro filas y siete columnas. Cada fila representa una semana en un almacén específico; cada columna representa un día en cada almacén. Los índices de las filas van de 0 a 3. Los índices de las columnas van de 0 a 6. La columna 0 es el lunes, la columna 1 es el martes, etc.

The diagram shows a 4x7 grid of numbers. To the left of the grid, a vertical double-headed arrow is labeled 'números de fila (almacenes)'. To the top of the grid, a horizontal double-headed arrow is labeled 'números de columna (días)'. The grid itself has row indices 0, 1, 2, 3 on the left and column indices 0, 1, 2, 3, 4, 5, 6 on top.

	0	1	2	3	4	5	6
0	22	49	4	93	0	12	32
1	3	8	67	51	5	3	63
2	14	8	23	14	5	23	16
3	54	0	76	31	4	3	99

Figura 15.1 Un arreglo bidimensional.

### PRÁCTICA DE AUTOEVALUACIÓN

**15.1** ¿Cuál columna representa el sábado? ¿Cuántas computadoras se vendieron el jueves en el almacén 3? ¿En cuál número de fila y de columna hallamos esa cifra?

## ● Declaración de matrices

Tal como hacemos con otras variables y objetos, para declarar una matriz utilizamos la palabra clave **new**, ya sea en la parte superior de la clase o de un método. El programador asigna un nombre a la matriz, como en los siguientes dos ejemplos:

```
int[,] ventas = new int[4, 7];
double[,] temps = new double[10, 24];
```

Al declarar una matriz hay que indicar su longitud en términos de filas y columnas. La matriz llamada **ventas** consta de cuatro filas, una para cada almacén; además, tiene siete columnas, una para cada día de la semana. Así, la matriz contiene cifras de ventas para cada uno de los cuatro almacenes durante cada día de la semana. Por su parte, la matriz **temps** contiene información sobre la temperatura de 10 hornos, para cada hora durante un periodo de 24 horas.

Al igual que con cualquier otra variable, en el caso de las matrices es común (y conveniente) elegir nombres que describan con claridad para qué se van a utilizar. Dicho nombre identificará la matriz completa, incluyendo toda la colección de datos que la conforma.

### PRÁCTICA DE AUTOEVALUACIÓN

**15.2** Declare una matriz para representar un tablero de ajedrez de  $8 \times 8$ . Cada "casilla" de la matriz debe contener una cadena de caracteres.

## ● Índices

Para hacer referencia a un elemento individual en una matriz bidimensional, el programa debe especificar los valores de dos índices enteros (también conocidos como subíndices). Por lo tanto, **ventas[3, 2]** se refiere al elemento que ocupa la posición determinada por la fila 3 y la columna 2

en la matriz; en nuestro ejemplo, esta referencia corresponde al almacén número 3 y al día número 2 (miércoles). De manera similar, `tableroAjedrez[2, 7]` podría contener la cadena de texto “peón”.

Podemos utilizar el siguiente código para dar un valor a uno de los elementos de la matriz:

```
ventas[2, 3] = Convert.ToInt32(textBox1.Text);
tableroAjedrez[3, 4] = textBox1.Text;
```

Además, es posible desplegar en cuadros de texto los valores de los elementos de la matriz de manera similar, y también podemos modificarlos con instrucciones de asignación, como en el siguiente ejemplo:

```
ventas[3, 2] = 99;
tableroAjedrez[2, 7] = "caballo"; // coloca la cadena de texto "caballo" en la
casilla especificada
```

Para referirnos a los elementos individuales de una matriz, en todos los fragmentos de código anteriores hemos especificado los valores de los índices que los representan.

Ahora bien, a menudo es necesario hacer referencia a un elemento de la matriz mediante la especificación de *variables* para cada uno de los dos índices. Ésta es la forma en que podemos aprovechar el poder de las matrices. Suponga que queremos sumar todos los números de una matriz que contiene datos sobre las ventas de computadoras en cuatro almacenes, durante un periodo de siete días:

```
int[, ] ventas = new int[4, 7];
```

La manera complicada de lograr nuestro propósito sería escribir lo siguiente:

```
suma =
    ventas[0, 0] + ventas[0, 1] + ventas[0, 2] + ventas[0, 3]
    + ventas[0, 4] + ventas[0, 5] + ventas[0, 6]
+ ventas[1, 0] + ventas[1, 1] + ventas[1, 2]
    + ventas[1, 3] + ventas[1, 4] + ventas[1, 5] + ventas[1, 6]
+ etcétera;
```

Este código es largo, difícil de entender y propenso a errores, pero es correcto. Sin embargo, no aprovecha la regularidad de las matrices. La alternativa sería utilizar un ciclo `for` en donde se emplean variables para guardar los valores de los índices. Al principio todos los índices se hacen igual a 0, y después su valor se incrementa en cada repetición del ciclo:

```
int[, ] ventas = new int[4, 7];
int suma;

suma = 0;
for ( int almacén = 0; almacén <= 3; almacén++ )
{
    for ( int númeroDía = 0; númeroDía <= 6; númeroDía++ )
    {
        suma = suma + ventas[almacén, númeroDía];
    }
}
```

Este código es mucho más corto y ordenado que si hubiéramos escrito todas las sumas con gran detalle.

### PRÁCTICA DE AUTOEVALUACIÓN

**15.3** Escriba instrucciones para colocar el texto “vacío” en cada uno de los cuadros del tablero de ajedrez.

## ● Longitud de la matriz

Cuando creamos una matriz de la siguiente forma:

```
double[,] info = new double[20, 40];
```

estamos dándole un tamaño fijo que no se puede modificar, a menos que volvamos a crear la matriz completa mediante `new`.

Por otro lado, tenemos oportunidad de averiguar la longitud de una matriz mediante el método `GetLength`. El argumento para este método especifica qué dimensión deseamos. Por ejemplo, para la matriz anterior podemos usar:

```
int tamañoFila;
tamañoFila = info.GetLength(0);
```

lo cual nos da un valor de 20 y:

```
int tamañoColumna;
tamañoColumna = info.GetLength(1);
```

nos da un valor de 40.

### PRÁCTICA DE AUTOEVALUACIÓN

**15.4** ¿Cuál es el valor de `tableroAjedrez.GetLength(0)`?

## ● Paso de matrices como parámetros

Suponga que necesitamos escribir un método cuya función sea calcular la suma de los elementos que conforman una matriz de enteros. Queremos que el método sea de propósito general, para poder manejar matrices de cualquier tamaño. Por lo tanto, debemos pasar el nombre de la matriz como parámetro del método; además, el resultado que se debe devolver al usuario del método es un número: la suma de los valores.

El siguiente es un ejemplo de una invocación al método:

```
int[,] ventas = new int[24, 12];
int total;
total = Suma(ventas);
```

Y el código del método es:

```
private int Suma(int[,] matriz)
{
    int total = 0;
    for ( int fila = 0; fila < matriz.GetLength(0); fila++ )
    {
        for ( int col = 0; col < matriz.GetLength(1); col++ )
        {
            total = total + matriz[fila, col];
        }
    }
    return total;
}
```

## ● Constantes

El uso de constantes puede evitar confusiones, en especial si dos matrices distintas tienen la misma longitud. Por ejemplo, en el programa para analizar la cantidad de ventas de computadoras en varios almacenes durante cierto número de días, utilizamos un arreglo bidimensional para almacenar las cifras. Cada columna representa un día; las filas albergan los datos de cada almacén. Ahora suponga que, por coincidencia, hay siete tiendas (es decir, el mismo número que la cantidad de días). La matriz sería:

```
int[, ] ventas = new int[7, 7];
```

El problema es que cada vez que veamos el dígito 7 en el programa, no sabremos si se refiere a la cantidad de almacenes o al número de días. Desde luego, en este caso eso no importa, ya que hay tantos almacenes como número de días; pero suponga que necesitamos modificar el programa de manera que trabaje con ocho almacenes. En ese caso sería muy conveniente poder usar el editor para cambiar cada ocurrencia del número 7 por el número 8. No obstante, eso constituye una opción muy peligrosa, ya que las longitudes son iguales.

Una excelente forma de mejorar el programa consistiría en declarar como constantes los valores máximos de los índices, como se muestra a continuación:

```
const int días = 7;
const int almacenes = 7;
```

Y después podríamos declarar la matriz así:

```
int[, ] ventas = new int[almacenes, días];
```

Ahora si cambia el número de almacenes podemos realizar con confianza la modificación correspondiente en el programa, ya que sólo tenemos que modificar un número en la declaración de las constantes. También podemos escribir ciclos `for` que hagan uso de las constantes:

```
for ( int índice = 0; índice < almacenes; índice++ )
{
    // cuerpo del ciclo
}
```



## ● Inicialización de matrices

Como sabemos, inicializar significa asignar un valor inicial a una variable. Si escribimos:

```
int[, ] tabla = new int[10, 10];
```

estamos preparando espacio en la memoria para la matriz, que contiene ceros. Además, el compilador asigna valores iniciales a las matrices que no se inicializan de manera explícita. Si la matriz consiste en números le asigna ceros; si consiste en cadenas de caracteres le asigna el valor "", y si consiste en objetos, asigna el valor `null` a todos sus elementos.

Una forma de inicializar una matriz de manera explícita es utilizar ciclos anidados, como en el siguiente ejemplo:

```
for ( int fila = 0; fila <= 9; fila++ )
{
    for ( int col = 0; col <= 9; col++ )
    {
        tabla[fila, col] = 99;
    }
}
```

Otra alternativa para inicializar una matriz es declararla como se muestra a continuación:

```
int[, ] tabla =
    {{1, 0, 1},
     {0, 1, 0}};
```

Observe el uso de las llaves y las comas. El código anterior crea una matriz con dos filas y tres columnas, y le asigna sus valores iniciales. Cuando utilizamos esta forma de inicialización, la longitud de la matriz *no* debe aparecer en los corchetes. La inicialización se lleva a cabo una vez, al momento de crear la matriz. Si el programa modifica el valor de uno de sus elementos, no será posible restaurarlo a su valor original, por lo menos hasta que vuelva a ejecutarse el programa.

En caso de que el programa necesite restablecer periódicamente la matriz de vuelta a sus valores iniciales, deberemos utilizar ciclos `for`, como vimos antes.

### PRÁCTICA DE AUTOEVALUACIÓN

**15.5** Escriba la declaración de una matriz de cadenas de texto con longitud de  $3 \times 3$ , de manera que se llene con las palabras uno, dos, tres, etc.

## ● Un programa de ejemplo

El siguiente programa mantiene un arreglo bidimensional (o matriz) de enteros. Los valores representan la precipitación pluvial en un periodo de siete días, en cada una de tres ubicaciones. La interfaz se muestra en la Figura 15.2. La matriz se despliega en un cuadro de texto multilínea, mostrando diversos valores iniciales. El usuario puede modificar un valor de la matriz, para lo cual debe especificar los índices del elemento en cuestión, y el nuevo valor de los datos.

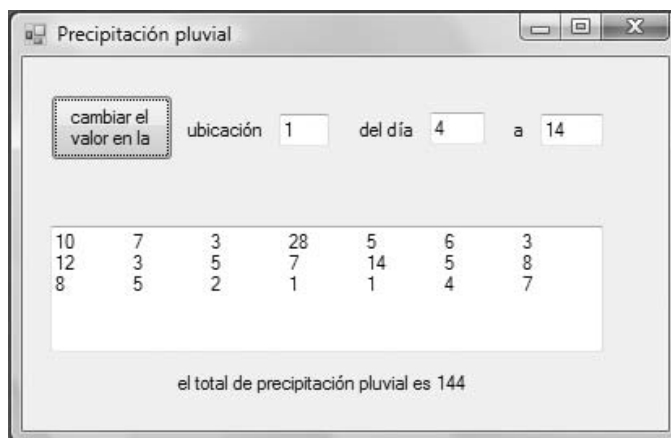


Figura 15.2 Arreglo bidimensional con datos sobre precipitación pluvial.

Primero declaramos la matriz:

```
int[,] datosLluvia =
    {{10, 7, 3, 28, 5, 6, 3},
     {12, 3, 5, 7, 12, 5, 8},
     { 8, 5, 2, 1, 1, 4, 7}};
```

Para desplegar todos los datos:

```
private void Mostrar()
{
    textBox1.Clear();
    for (int ubicación = 0; ubicación <= 2; ubicación++)
    {
        for (int númeroDía = 0; númeroDía <= 6; númeroDía++)
        {
            textBox1.AppendText(Convert.ToString
                                (datosLluvia[ubicación, númeroDía])
                                + "\t");
        }
        textBox1.AppendText("\r\n");
    }
}
```

El ciclo `for` interior recorre los distintos días, mientras que el ciclo `for` exterior recorre las distintas ubicaciones.

Para cambiar un elemento de la matriz hay que obtener el número del día, el número de la ubicación y el nuevo valor de datos, para lo cual se emplean los cuadros de texto respectivos:

```
private void CambiarValor()
{
    int valorDatos;
    int númeroDía;
    int ubicación;

    númeroDía = Convert.ToInt32(díaTextBox.Text);
    ubicación = Convert.ToInt32(ubicaciónTextBox.Text);
    valorDatos = Convert.ToInt32(valorTextBox.Text);
    datosLluvia[ubicación, númeroDía] = valorDatos;

    Mostrar();
    CalcularTotal();
}
```

Para calcular el total de precipitación pluvial en todas las ubicaciones:

```
private void CalcularTotal()
{
    int total = 0;

    for (int ubicación = 0; ubicación <= 2; ubicación++)
    {
        for (int númeroDía = 0; númeroDía <= 6; númeroDía++)
        {
            total = total + datosLluvia[ubicación, númeroDía];
        }
    }

    label1.Text = "el total de precipitación pluvial es " + Convert.ToString(total);
}
```

Al ejecutar este programa tenga cuidado de introducir los números de fila en el rango de 0 a 2, y los números de columna en el rango de 0 a 6.

En este código podemos ver de nuevo, que es muy común encontrar instrucciones `for` anidadas con arreglos bidimensionales, ya que aprovechan al máximo la uniformidad de éstas.

## Fundamentos de programación

Los arreglos bidimensionales, o matrices, son una colección de datos con un solo nombre (por ejemplo, `datosLluvia`). La matriz se puede comparar con una tabla de dos dimensiones: filas y columnas. Suponga que queremos representar los datos de precipitación pluvial para cada uno de los siete días de la semana en tres lugares. En ese caso declaramos la siguiente matriz:

```
int[, ] datosLluvia = new int[7, 3];
```

Para acceder a sus elementos especificamos dos índices, que deben ser números enteros; por ejemplo, `datosLluvia[4, 2]`. El primer índice describe el número de fila, y el segundo el número de columna. Al crear la matriz se especifica la longitud de sus dos dimensiones: 7 y 3, en este ejemplo. Nuestra matriz tiene siete filas y tres columnas. Dado que los índices siempre empiezan en 0, en este caso los índices de las filas van de 0 a 6, y los índices de las columnas de 0 a 2.

Los elementos de la matriz pueden ser de cualquier tipo: `int`, `double`, `string` o cualquier otro objeto, pero dentro de una misma matriz todos deben ser del mismo tipo (`int`, en nuestro ejemplo). La excepción es cuando se declara una matriz que consiste en objetos tipo `object`. En este caso la matriz puede dar cabida a cualquier mezcla de objetos.

Es común utilizar ciclos `for` anidados junto con los arreglos bidimensionales.

En este libro exploramos tanto los arreglos unidimensionales como los bidimensionales. C# puede trabajar con matrices de hasta 60 dimensiones, pero en la práctica es muy raro utilizar más de 3 dimensiones.

## Errores comunes de programación

Un error común en C# consiste en confundir la longitud de una matriz con el rango de índices válidos. Por ejemplo, la matriz:

```
int[, ] tabla = new int[11, 6];
```

consta de 11 filas y 6 columnas. El rango válido de los índices para las filas es de 0 a 10; el rango válido de índices para las columnas es de 0 a 5. La referencia a `tabla[11, 6]` hará que el programa se detenga y aparezca un mensaje de error, dado que los valores de fila y columna están fuera de rango.

## Resumen

- Los arreglos bidimensionales o matrices, son colecciones de datos organizadas en forma de tabla, con filas y columnas.
- El programador debe asignar el nombre al momento de crear la matriz.
- Las matrices se declaran (al igual que otras variables) de la siguiente forma:

```
int[,] animales = new int[25, 30];
```

en donde 25 es el número de filas y 30 el número de columnas.

- Para hacer referencia a un elemento individual de una matriz utilizamos índices enteros, como en el siguiente ejemplo:

```
animales[12, 3] = 45;
```

## EJERCICIOS

## Operaciones básicas con matrices

**15.1 Manejador de datos** Escriba un programa que utilice una matriz de longitud  $4 \times 7$ , de números enteros, similar a la que usamos en Precipitación pluvial (con los resultados que se muestran en la Figura 15.2). Amplíe el programa para que contenga los siguientes métodos:

- Al oprimir un botón etiquetado como “sumas”, sumar los valores de cada una de las siete columnas y todos los valores de cada una de las cuatro filas; los resultados deberán desplegarse en pantalla.
- Al oprimir un botón etiquetado como “mayor”, encontrar el valor más grande de cada fila, de cada columna, y de toda la matriz.
- Al oprimir un botón etiquetado como “escalar”, multiplicar cada uno de los valores que conforman la matriz por un número introducido por el usuario en un cuadro de texto (esto podría utilizarse para hacer conversiones de centímetros a pulgadas).

## Medidas estadísticas

**15.2** Amplíe el programa de precipitación pluvial de manera que proporcione un botón para calcular el promedio diario de precipitación pluvial para cada ubicación. Por ejemplo, la precipitación pluvial promedio diaria en la ubicación dos podría ser 23.

Mejore todavía más el programa para que tenga un botón que calcule la media y la desviación estándar de la precipitación pluvial diaria en cualquier ubicación. Por ejemplo, la media de la precipitación pluvial de cualquier ubicación podría ser de 19, con una desviación estándar de 6.4.

## Gráficas de barras y circulares

**15.3** Enriquezca el programa Precipitación pluvial de manera que el usuario pueda especificar una fila (ubicación). Después deberá desplegarse la información correspondiente mediante una gráfica de barras.

Extienda el programa para que muestre la información de una sola fila o columna como gráfica circular. Utilice para ello el método de biblioteca `FillPie`.

## Operaciones matemáticas

**15.4 Transpuesta** “Transpuesta de una matriz” es el término técnico que se utiliza para describir la acción de intercambiar diagonalmente los elementos de la matriz, es decir, sus columnas por sus filas, o viceversa. Los números en la diagonal principal no cambian. Por ejemplo, si tenemos la matriz:

1	2	3	4
6	7	8	9
10	11	12	13
14	15	16	17

su transpuesta es:

1	6	10	14
2	7	11	15
3	8	12	16
4	9	13	17

Escriba un programa que reciba como entrada los elementos de una matriz de manera similar al programa Precipitación pluvial. Cuando el usuario haga clic en un botón, deberá realizarse la transposición de la matriz, desplegando el resultado en pantalla.

## Juegos

**15.5 Gato** El juego conocido como “gato” o “tres en línea” se practica en una cuadrícula de  $3 \times 3$ , que en un principio está vacía. Participan dos jugadores, por turnos. Uno de ellos coloca una cruz en un cuadro vacío, y el otro coloca un círculo en otro.

Gana la persona que obtenga una línea de tres círculos o tres cruces. Por ejemplo, una victoria para el participante que usa círculos podría ser:

o	x	o
x	o	
x		o

La partida puede terminar en empate si ninguno de los jugadores logra hacer una línea de tres.

Escriba un programa para jugar “gato”. Debe haber sólo un botón para iniciar un nuevo juego. El programa mostrará los círculos y las cruces de manera gráfica, cada uno en su propio cuadro de imagen. Para especificar un movimiento, el jugador humano debe hacer clic con el ratón en el cuadro de imagen en donde se vaya a colocar la cruz. El otro jugador será la computadora, que decidirá al azar en dónde colocar los círculos.

## Vida artificial

**15.6 Vida celular** Un organismo consiste en células individuales que están encendidas (vivas) o apagadas (muertas). Cada generación de vida consiste en una sola fila de células, y cada una de dichas filas depende de la anterior (al igual que en la vida real). El tiempo se desplaza de arriba hacia abajo. Cada fila representa una generación. Las vidas se ven así:

				*					
				*	*	*			
			*	*		*			
		*	*		*	*	*	*	
	*	*			*			*	
*	*		*	*	*	*	*	*	*

Al principio sólo hay una célula viva. El que una célula esté viva o muerta depende de una combinación de factores: si estaba viva o muerta en la última generación, y si sus vecinas

inmediatas estaban vivas o muertas en la última generación. Podemos ver que, aunque sólo han “transcurrido” cinco generaciones, está surgiendo un patrón. Estos patrones son muy sutiles, e imitan a los que se encuentran en los organismos vivos de la vida real. Las reglas son las siguientes:

Una célula sólo vive si:

- estaba muerta pero sólo su vecina izquierda estaba viva;
- estaba muerta pero sólo su vecina derecha estaba viva;
- estaba viva pero sus vecinas inmediatas estaban muertas;
- estaba viva y sólo su vecina derecha estaba viva;

Por ejemplo, dada la siguiente generación:

	*	*	*	
--	---	---	---	--

- La primera célula vive, ya que aunque estaba muerta, su vecina inmediata a la derecha estaba viva.
- La segunda célula vive, ya que sólo su vecina derecha inmediata estaba viva.
- La tercera célula viva muere (suponemos que debido a la sobrepoblación).
- La cuarta célula muere.
- La quinta célula vive ya que, aunque estaba muerta, su vecina izquierda inmediata estaba viva.

Por lo tanto, la nueva generación es:

*	*			*
---	---	--	--	---

Escriba un programa que utilice un arreglo bidimensional para graficar el progreso de esta forma de vida. Despliegue el desarrollo en pantalla, utilizando asteriscos para representar los organismos, como en las figuras anteriores. Agregue un botón que permita al usuario avanzar a la siguiente generación.

**15.7 El Juego de la vida de Conway** En esta forma de vida, un organismo consiste en células individuales que están encendidas (vivas) o apagadas (muertas). Los organismos existen en un mundo representado por una cuadrícula bidimensional, como en esta figura:

		*		*			
	*		*	*			
			*		*		
		*					
	*						

Las reglas que gobiernan este organismo son:

- Si una célula viva tiene dos o tres vecinas, sobrevivirá. En caso contrario morirá por aislamiento o sobrepoblación.

- Si una célula vacía está rodeada por exactamente tres células, nacerá una nueva célula viva para llenar el espacio.
- Todos los nacimientos y muertes ocurren de manera simultánea.

Escriba un programa para simular este tipo de vida. Al principio el programa deberá permitir que el usuario haga clic en las células que vayan a estar vivas. Agregue un botón que permita avanzar a la siguiente generación y desplegarla en pantalla.

El programa requiere dos matrices: una para representar el estado actual de vida, y otra para representar la siguiente generación. Después de crear cada nueva generación se intercambiarán las funciones de las dos matrices.

#### **SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**

---

**15.1** La columna 5 es sábado. Se vendieron 31 computadoras en el almacén 3 el jueves. Ésta es la fila 3 y columna 3.

**15.2**     `string[, ] tableroAjedrez = new string[8, 8];`

**15.3**     `string[, ] tableroAjedrez = new string[8, 8];`

```
for ( int fila = 0; fila <= 7; fila++ )
{
    for ( int col = 0; col <= 7; col++ )
    {
        tableroAjedrez[fila, col] = "vacío";
    }
}
```

**15.4** El valor (y por lo tanto, la longitud) es 8.

```
15.5     string[,] números =
           {
               {"uno", "dos", "tres"},
               {"cuatro", "cinco", "seis"},
               {"siete", "ocho", "nueve"}
           };
```



# Manipulación de cadenas de caracteres

## En este capítulo conoceremos:

- las herramientas para trabajar con cadenas de caracteres que hemos utilizado hasta ahora;
- los principales métodos y propiedades de la clase `String`;
- las expresiones regulares.

## ● Introducción

Las cadenas de caracteres son muy importantes en el desarrollo de software. Todos los lenguajes de programación tienen herramientas para la manipulación primitiva de caracteres, pero C# cuenta con una colección de métodos especialmente útil. En este capítulo analizaremos todas las herramientas para trabajar con cadenas que hemos utilizado hasta este momento, y ampliaremos nuestro conocimiento al estudiar el conjunto de métodos disponibles para el procesamiento de cadenas.

Las siguientes son algunas situaciones en las que se utilizan cadenas de caracteres:

- Para desplegar mensajes en pantalla (quizá mediante la colocación de texto en etiquetas).
- Para recibir texto del usuario. A menudo esto se hace mediante un cuadro de texto, pero en una aplicación de consola el usuario podría introducir texto a través de la línea de comando.
- Para almacenar datos en archivos. Cuando trabajemos con archivos en el capítulo 18, veremos que el contenido de muchos tipos de archivo puede considerarse como una secuencia de cadenas. Además, los nombres de los archivos y las carpetas son cadenas de caracteres.
- Para realizar búsquedas en páginas Web.
- Para guardar texto en memoria, de manera que pueda ser empleado por los procesadores y editores de texto.

## ● Uso de cadenas de caracteres: un recordatorio

En esta sección hablaremos sobre las herramientas para trabajar con cadenas de caracteres que hemos visto hasta ahora.

Podemos declarar variables y proveer un valor inicial, como en el siguiente ejemplo:

```
string x;  
string y = "México";
```

También podemos asignar una cadena a otra, como en:

```
x = "Inglaterra";  
y = "Francia";  
y = x;  
x = "";      // una cadena de longitud cero
```

Esta última instrucción ilustra que la longitud de una cadena puede variar. En sentido estricto, la cadena anterior se destruye y es sustituida por un valor completamente nuevo. El espacio que ocupaba la cadena quedará disponible para que otras variables lo utilicen.

Además, podemos usar el operador + para concatenar cadenas de caracteres, como en el siguiente ejemplo:

```
MessageBox.Show("Vivo en " + y);
```

Un procedimiento muy utilizado en el procesamiento de cadenas es empezar con una cadena vacía y unirle elementos a medida que se ejecuta el programa. Para ello podríamos utilizar la siguiente instrucción:

```
x = x + "algo";
```

que agrega texto al final de **x**. A esto se le conoce como “adjuntar”.

Asimismo, podemos comparar cadenas con los signos == y !=, como en:

```
if (x == "Francia")  
{  
    //hacer algo  
}
```

Y crear arreglos de cadenas de caracteres (con índices que empiezan en 0), como en el siguiente ejemplo:

```
string [] ciudades = new string[10]; // 0 a 9
```

Por último, podemos convertir cadenas de caracteres a números, y viceversa; para ello empleamos los métodos de la clase **Convert**. Esto es útil cuando recibimos cadenas a partir de un cuadro de texto (o de un archivo, como veremos más adelante). Por ejemplo, podríamos utilizar el siguiente código:

```
int n = 3;  
double d;  
x = Convert.ToString(n);  
y = "123";  
n = Convert.ToInt32(y);  
d = Convert.ToDouble("12.345");
```

Si una cadena no se puede convertir en número se produce una excepción. En el capítulo 17 veremos cómo interceptar y manejar excepciones.

Esto es lo que hemos visto hasta el momento. A continuación analizaremos detalles sobre las cadenas de caracteres y los métodos disponibles.

## ● Indexación de cadenas

Cada uno de los caracteres que conforman una cadena de caracteres tiene un índice numérico, empezando desde cero. Pero cuidado: es fácil confundir la longitud de una cadena con el valor máximo del índice. Para comprender esta situación veamos el siguiente diagrama, que muestra la cadena **posible**:

Caracteres:	p	o	s	i	b	l	e
Índices:	0	1	2	3	4	5	6

Como puede ver, el valor máximo del índice es 6 y la longitud de la cadena (el número de caracteres que contiene) es 7.

## ● Caracteres dentro de las cadenas de caracteres

Las cadenas pueden contener todos los caracteres que queramos, pero al crear valores de cadena entre comillas hay varios casos especiales.

El carácter `\` tiene un significado especial cuando está entre comillas; se le denomina carácter de *escape*: el carácter que le sigue recibe un tratamiento especial. He aquí las situaciones principales:

- `\r` y `\n` representan los caracteres de *retorno de línea* y *nueva línea*. Ambos son utilizados para marcar el final de una línea, tanto en cuadros de texto como en archivos.
- `\"` representa el carácter comilla. Esto nos permite crear cadenas de caracteres que contienen comillas, como en el siguiente ejemplo:

```
textBox1.Text = "La palabra \"objeto\"";
```

que desplegaría lo siguiente:

```
La palabra "objeto"
```

- El uso de `@` para facilitar el uso del carácter `'\'` en una cadena de caracteres. La utilización más común de este carácter es en las rutas de archivos de Windows. Históricamente, el signo `'\'` como carácter de escape proviene del lenguaje C. Más adelante Microsoft adoptó el mismo carácter como separador de rutas de archivos para Windows, y aquí aparece un problema. Utilizamos `@` para permitir que el carácter `\` valga por sí mismo, como en:

```
string miArchivo = @"c:\notas\cosasCSharp\Lista.txt";
```

Sin `@`, C# consideraría cada uno de los caracteres situados después de `\` como algo especial.

- El uso de `@` también puede simplificar la escritura de una cadena que contiene varias líneas, como en el siguiente ejemplo:

```
|           textBox1.Text = @"He aquí una cadena:
|está dividida
|en tres líneas.";
```

El cuadro de texto se desplegaría así:

```
He aquí una cadena:
está dividida
en tres líneas.
```

Por lo general nuestros fragmentos de código no muestran su sangría completa, pero en el ejemplo anterior representamos con una línea el margen izquierdo para indicarle que, si usted aplicara sangría a las últimas dos líneas, éstas aparecerían anteceditas por espacios adicionales en el cuadro de texto. Cualquier carácter de espacio y de fin de línea se mantiene en la cadena final.

El uso de @ es sólo por conveniencia, pero puede mejorar la legibilidad de sus códigos.

### ● Una observación sobre el tipo `char`

---

En este libro es más importante la claridad de un programa que su velocidad de ejecución. Sin embargo, algunas veces enfrentaremos situaciones en las que podría ser conveniente evitar el “castigo” que constituye el tiempo adicional implícito en el uso de cadenas de caracteres. Considere, por ejemplo, el caso en que nuestras cadenas sólo contienen un carácter. C# cuenta con un tipo de datos llamado `char`, el cual almacena cada elemento como carácter Unicode de 16 bits; en consecuencia, podemos realizar comparaciones entre valores `char` con la misma rapidez que con enteros. He aquí algunos ejemplos:

```
char inicial = 'M';
char marcador = '\n';
char letra;

letra=inicial;
if (letra== 'B')
{
    // etc...
}
```

Observe el uso de la comilla sencilla en vez de la comilla doble que utilizamos para las cadenas de caracteres. Estas comillas sólo pueden contener un carácter (aunque parezca que `\n` consta de dos caracteres, se reemplaza en tiempo de compilación por el carácter individual de nueva línea).

Para hacer conversiones entre `char` y `string` podemos usar esta instrucción:

```
inicial = Convert.ToChar("x");
string letraComoCadena = Convert.ToString(letra);
```

El tipo `char` se utiliza muy poco, pero algunas veces es necesario emplearlo con los métodos de biblioteca (como `split`, del cual hablaremos más adelante).

### ● Métodos y propiedades de la clase `String`

---

En esta sección examinaremos los métodos más útiles de la clase `String` (observe la S mayúscula, que indica que se trata de una clase). (C# cuenta con la palabra clave `string` como un alias para la clase `String`. Por eso utilizamos el término `string`.) El siguiente marco de trabajo le servirá para utilizar estos métodos:

```
private void button1_Click(object sender, EventArgs e)
{
    string cadena1;
    string cadena2;
    string cadenaResultante;
```

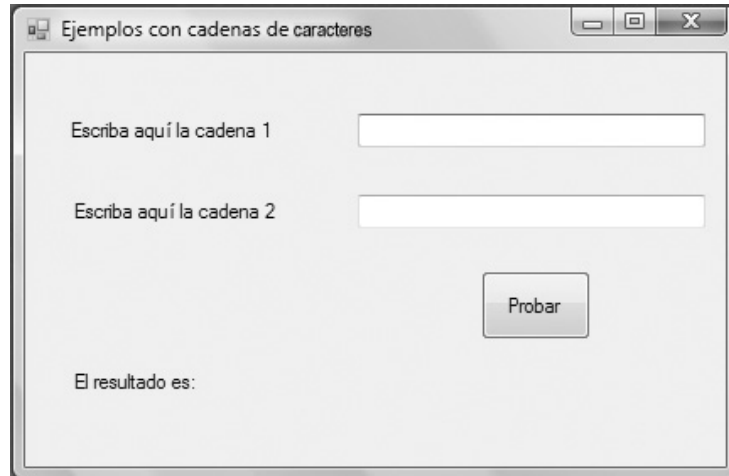


Figura 16.1 Interfaz del marco de trabajo para los ejemplos de cadenas.

```
int n, m;  
string [] palabras;  
//coloque aquí el código de ejemplo:  
  
//fin del ejemplo  
}
```

La interfaz de este marco de trabajo se muestra en la Figura 16.1. El programa le proporciona dos cuadros de texto (`cadena1TextBox` y `cadena2TextBox`) y una etiqueta (`resultadoLabel`) para desplegar las respuestas. Podrá escribir las secciones de ejemplo que veremos a continuación en el espacio indicado por la línea:

```
// coloque aquí el código de ejemplo
```

## ● Comparación de cadenas

Recuerde que podemos comparar cadenas mediante los operadores `==` y `!=`. Sin embargo, a menudo necesitamos colocar las cadenas en orden alfabético, para lo cual debemos poder determinar si una cadena va antes que otra. Ahora bien, antes mencionamos que cada carácter tiene un código numérico propio. Imagine que `a` es el menor, `b` es el siguiente, etcétera. Después de `z` tenemos `A`, `B`, etcétera. Las cadenas se comparan empezando desde el carácter ubicado en el extremo izquierdo. He aquí algunos ejemplos:

```
antena va antes que barco  
ancho va antes que antenna  
algo va antes que algoritmo  
ANTENA va antes que BARCO  
antena va antes que INSECTO  
insecto va antes que INSECTO
```

El método `compareTo` devuelve un resultado entero con el siguiente significado:

- 0 si las cadenas son iguales;
- -1 si el objeto cadena va antes que el parámetro;
- 1 si el objeto cadena va después que el parámetro.

Si utilizamos:

```
n = "antena".CompareTo("barco");
```

`n` recibirá un `-1`. Podemos comprobar este resultado con el siguiente código:

```
//ejemplo de código:
MessageBox.Show("Escriba la cadena principal y la cadena " +
    " con la que se va a comparar.");
cadena1 = cadena1TextBox.Text;
cadena2 = cadena2TextBox.Text;
if (cadena1.CompareTo(cadena2) == -1)
{
    resultadoLabel.Text = cadena1 + " va antes que " + cadena2;
}
else
    if (cadena1.CompareTo(cadena2) == 1)
    {
        resultadoLabel.Text = cadena1 + " va después que " + cadena2;
    }
else
    {
        resultadoLabel.Text = "Las cadenas son iguales.";
    }
```

## ● Corrección de cadenas

---

En esta sección veremos métodos para modificar cadenas de caracteres. Estos métodos en realidad crean una nueva cadena, en vez de modificar la original.

### **ToLower**

El método `ToLower` convierte cualquier letra mayúscula que se encuentre en una cadena en letra minúscula. En el siguiente ejemplo:

```
cadena1 = "Versión 1.1";
cadenaResultante = cadena1.ToLower();
```

coloca la cadena `"versión 1.1"` en `cadenaResultante`. Utilice estas líneas de código para experimentar con su propio programa:

```
//código de ejemplo:
cadena1 = cadena1TextBox.Text;
resultadoLabel.Text = cadena1.ToLower();
```

### ToUpper

El método `ToUpper` realiza una operación similar a la de `ToLower`, pero cambia las letras minúsculas por sus equivalentes en mayúscula. Por ejemplo:

```
cadena1 = "C Sharp";
cadenaResultante = cadena1.ToUpper();
```

El resultado sería que `cadenaResultante` tomaría el valor `"C SHARP"`.

### Trim

El método `Trim` elimina espacios a ambos extremos de una cadena. Si utilizamos:

```
cadena1 = "          Centro          ";
cadenaResultante = cadena1.Trim();
```

`cadenaResultante` se convertirá en `"Centro"`. El código de ejemplo que puede emplear para lograrlo es:

```
//código de ejemplo:
cadena1 = cadena1TextBox.Text;
resultadoLabel.Text = cadena1.Trim();
```

### Insert

Este método nos permite insertar caracteres en una posición específica de la cadena, como en:

```
cadena1 = "Programación en C Sharp";
cadenaResultante = cadena1.Insert(16, "Visual ");
```

El resultado es `"Programación en Visual C Sharp"`. Aquí insertamos una cadena en el carácter número 16 ("C"). Los caracteres subsecuentes se desplazan a la derecha para hacer espacio. He aquí el código de ejemplo que puede utilizar:

```
//ejemplo de código:
MessageBox.Show("Escriba la cadena a insertar, y " +
    "especifique la posición en la que se va a colocar");
cadena1 = cadena1TextBox.Text;
n = Convert.ToInt32(cadena2TextBox.Text);
cadena2 = "Una cadena a insertar en...";
resultadoLabel.Text = cadena2.Insert(n, cadena1);
```

### Remove

Este método elimina cierto número de caracteres de una posición específica, como en:

```
cadena1 = "Fecha final";
cadenaResultante = cadena1.Remove(1, 6);
```

El orden de los argumentos es: primero la posición inicial, seguida del número de caracteres a eliminar. El resultado es "Final", ya que se eliminan seis caracteres empezando en la posición 1 (se eliminaron los caracteres "echa f"). El código de ejemplo es:

```
//ejemplo de código:
MessageBox.Show("Especifique la posición inicial, y " +
    "el número de caracteres a eliminar");
m = Convert.ToInt32(cadena1TextBox.Text);
n = Convert.ToInt32(cadena2TextBox.Text);
cadena1 = "Un ejemplo de cómo eliminar caracteres de una cadena...";
resultadoLabel.Text = cadena1.Remove(m, n);
```

## ● Análisis de cadenas

---

Los métodos y propiedades que comentaremos a continuación nos permiten examinar una cadena de caracteres para, por ejemplo, extraer una sección (o *subcadena*) de la misma.

### Length

La propiedad `Length` proporciona el número de caracteres que conforman la cadena de caracteres, como en:

```
cadena1 = "Programación en C#";
cadenaResultante = Convert.ToString(cadena1.Length);
```

En este caso el resultado es 18.

### Substring

El método `Substring` copia una parte específica de la cadena. Debemos proveer la posición inicial y el número de caracteres a copiar. Por ejemplo:

```
cadena1 = "posición";
cadenaResultante = cadena1.Substring(5, 3);
```

En este caso seleccionamos desde la letra *i* (que está en el índice 5) un total de 3 caracteres, de manera que el resultado es *ión*. El valor de `cadena1` no se modifica.

Éste es el código para el programa de ejemplo.

```
//ejemplo de código:
MessageBox.Show("Especifique la posición inicial y " +
    "el número de caracteres a copiar");
m = Convert.ToInt32(cadena1TextBox.Text);
n = Convert.ToInt32(cadena2TextBox.Text);
cadena1 = "Una cadena de ejemplo, de la que se extrae una subcadena";
resultadoLabel.Text = cadena1.Substring(m, n);
```



Una aplicación común de este método es para obtener un solo carácter de una cadena, como en el siguiente ejemplo:

```
n = 3;
cadena1 = cadena2.Substring(n, 1);
```

Al variar la `n` en un ciclo podemos extraer cada carácter de una cadena, uno a la vez.

#### PRÁCTICA DE AUTOEVALUACIÓN

**16.1** Explique el efecto del siguiente código:

```
string palabra = "posible";
string s = palabra.Substring(1, palabra.Length-2);
```

#### IndexOf

Este método determina si una subcadena está contenida en una cadena. Además, nos permite proporcionar un desplazamiento para especificar en dónde debe empezar la búsqueda. Por ejemplo:

```
cadena1 = "mississippi!";
n = cadena1.IndexOf("si");
```

En este caso la `n` toma el valor 3 para mostrar la posición de la primera ocurrencia de `si` (recuerde que la primera posición de una cadena lleva el índice 0).

Pero si utilizamos:

```
cadena1 = "mississippi!";
n = cadena1.IndexOf("si", 5);
```

`n` se convierte en 6, ya que se ignora la primera ocurrencia. Si no se encuentra la cadena, el método devuelve `-1`.

He aquí el código para el programa de ejemplo, que informa si una cadena contiene una subcadena o no.

```
//ejemplo de código:
MessageBox.Show("Escriba la cadena principal y la subcadena");
cadena1 = cadena1TextBox.Text;
cadena2 = cadena2TextBox.Text;
if (cadena1.IndexOf(cadena2) == -1)
{
    resultadoLabel.Text = "¡cadena no encontrada!";
}
else
{
    resultadoLabel.Text = "cadena encontrada";
}
```

### Split

Este método nos ayuda a dividir una cadena en secciones. He aquí un ejemplo:

```
string[] palabras;
char[] separadores = {' ',' '};
cadena1 = "Guitarra, bajo, batería";
palabras = cadena1.Split(separadores);
for (int lugar = 0; lugar < palabras.Length; lugar++)
{
    palabras[lugar] = palabras[lugar].Trim();
}
```

En este ejemplo utilizamos un arreglo de tipo `char`, el cual puede contener varios separadores alternativos. En este caso no los hay: se utiliza una coma como el único separador entre los tres elementos, pero hemos aumentado el realismo en los datos al agregarles espacios adicionales. El método `split` devuelve un arreglo de cadenas de tamaño apropiado (que, por supuesto, no conocemos de antemano). En el ejemplo anterior podemos ver que el arreglo de cadenas se declaró sin un tamaño, lo cual es inusual, pero está permitido. Después la utilizamos para guardar el resultado del método `split`, y en este punto podemos emplear la propiedad `Length` del arreglo para controlar un ciclo que procese cada elemento por separado. Aquí el procesamiento implica eliminar los espacios adicionales. Antes de esa eliminación, los valores del arreglo son:

```
"Guitarra"
"  bajo  "
"  batería"
```

Tras la eliminación de espacios tenemos:

```
"Guitarra"
"bajo"
"batería"
```

### LastIndexOf

Este método es similar, en concepto, a `IndexOf`, sólo que devuelve la posición de la ocurrencia de la subcadena que esté más cerca del extremo derecho. Se devuelve el valor `-1` si no hay una coincidencia. He aquí un ejemplo:

```
cadena1 = @"directorio\archivo";
n = cadena1.LastIndexOf(@"\");
```

El valor devuelto es 11.

### StartsWith

Este método se utiliza para averiguar si una cadena empieza con una subcadena específica. Su ventaja es que evita la complejidad adicional de utilizar la propiedad `Length`. El método devuelve un valor booleano. Por ejemplo:

```
cadena1 = "http://ruta/pagina.html";
resultadoInicia = cadena1.StartsWith("http");
```

En este caso `resultadoInicia` quedaría como verdadero.

He aquí el código para el programa de ejemplo, que determina si una subcadena está presente al inicio de una cadena:

```
//ejemplo de código:
MessageBox.Show("Escriba la cadena principal y la cadena" +
    "que debe buscarse en el principio");
cadena1 = cadena1TextBox.Text;
cadena2 = cadena2TextBox.Text;
if (cadena1.StartsWith(cadena2))
{
    resultadoLabel.Text = "La cadena buscada se encontró al principio";
}
else
{
    resultadoLabel.Text = "La cadena buscada no se encontró al principio";
}
```

### EndsWith

Este método se utiliza para averiguar si una cadena termina con una subcadena específica. Aunque podemos utilizar una combinación de otros métodos para lograr el mismo resultado, `EndsWith` es menos propenso a errores. El método devuelve un valor booleano. Por ejemplo:

```
cadena1 = "http://ruta/pagina.html";
resultadoTermina = cadena1.EndsWith("html");
```

En este caso `resultadoTermina` quedaría como verdadero.

He aquí el código del programa de ejemplo, el cual determina si una subcadena está presente al final de una cadena:

```
//ejemplo de código:
MessageBox.Show("Escriba la cadena principal y la cadena" +
    "que debe buscarse al final");
cadena1 = cadena1TextBox.Text;
cadena2 = cadena2TextBox.Text;
if (cadena1.EndsWith(cadena2))
{
    resultadoLabel.Text = "La cadena buscada se encontró al final";
}
else
{
    resultadoLabel.Text = "La cadena buscada no se encontró al final";
}
```

## ● Expresiones regulares

---

Casi toda la programación con cadenas de caracteres puede llevarse a cabo con las herramientas que hemos visto hasta ahora, de manera que si es la primera vez que lee este capítulo y quiere pasar por alto esta sección, hágalo.

Sin embargo, a veces surge la necesidad de realizar complicadas comprobaciones en los caracteres de una cadena, tal vez para validar alguna entrada del usuario. Ciertamente podemos usar los métodos para trabajar con cadenas que vimos antes, pero también tenemos la alternativa de emplear *expresiones regulares*. He aquí algunos ejemplos de pruebas complicadas:

- El código de un producto debe consistir en una letra mayúscula seguida de un dígito.
- El código del personal debe consistir en cualquier cantidad de dígitos.
- La calificación de un estudiante debe consistir en **A**, **B** o **C** por su logro, seguida de **1**, **2** o **3** por su esfuerzo.

Las expresiones regulares nos permiten construir un patrón con el cual comparar nuestros datos.

Las cadenas de expresión regular contienen símbolos especiales que describen las posibilidades de búsqueda para cada carácter. He aquí una lista de los más comunes:

Símbolo	Acción
.	Busca coincidencias con cualquier carácter
^	Busca coincidencias con el inicio de una cadena
\$	Busca coincidencias con el final de una cadena
[ ]	Encierra las alternativas para búsquedas para un solo carácter. Por ejemplo, <code>[ABC]</code> indica que un carácter puede ser A, B o C
{ }	Encierra una cuenta repetitiva. Por ejemplo, <code>{5}</code> busca una coincidencia exacta con cinco ocurrencias del elemento anterior
+	Busca coincidencias con una o más ocurrencias del elemento anterior
-	Expresa un rango de caracteres. Por ejemplo, <code>A-Z</code> busca coincidencias con cualquier letra mayúscula
\	Provee al siguiente carácter su significado normal. Por ejemplo, <code>\.</code> buscará coincidencias con el carácter <code>.</code> en vez de buscarlas con cualquier carácter

### Uso del método `IsMatch`

Las expresiones regulares tienen varios usos; en esta sección nos concentraremos en comprobar que el contenido de una cadena cumpla con nuestros requerimientos.

El método `IsMatch` (un método estático de la clase `Regex`) nos permite comprobar las coincidencias entre una cadena de datos y una cadena de expresión regular. El valor que devuelve es `true` o `false`, dependiendo de si se encontró una coincidencia o no. Para utilizar este método debemos especificar un nuevo espacio de nombres en la parte superior de nuestro programa:

```
using System.Text.RegularExpressions;
```

He aquí un ejemplo:

```
string calif = "b2";
if (Regex.IsMatch(calif, "[ABC][123]$"))
{
    resultadoLabel.Text = "sí";
}
else
{
    resultadoLabel.Text = "no";
}
```

Es necesario que proporcionemos dos parámetros a `IsMatch`: nuestra cadena de datos y una cadena de expresión regular. En el ejemplo anterior la expresión regular especifica dos caracteres. El primero puede ser `A`, `B` o `C`; el segundo puede ser `1`, `2` o `3`. De hecho la coincidencia se valorará como falsa, ya que la `b` de nuestra cadena de datos está en minúscula. Sin embargo, la cadena `"B2"` sí produciría una coincidencia.

Cabe mencionar que `IsMatch` explora a lo largo de la cadena, de manera que si especificáramos `"marcar B2 datos"` habría una coincidencia, debido al par de caracteres que están en medio de la cadena. Para evitar esto podemos utilizar los signos `^` y `$`, mediante los cuales especificamos el inicio y fin de la cadena; la sintaxis sería ésta: `"^[ABC][123]$"`.

He aquí algunas expresiones regulares y las cadenas con las que buscan coincidencias:

Expresión	Coincidencias buscadas
"^.. \$"	Busca coincidencias con dos caracteres cualesquiera, como <code>AD</code> , <code>45</code> , <code>//</code>
"^..A \$"	Busca coincidencias con cualquier carácter seguido de una <code>A</code>
"^A-Z \$"	Busca coincidencias con cualquier letra mayúscula
"^A-Za-z +"	Busca coincidencias con una letra mayúscula seguida de una o más letras minúsculas
"^[0123456789]{5} \$"	Busca coincidencias con exactamente cinco dígitos

Las expresiones regulares pueden ser muy útiles, pero si son demasiado largas podrían llegar a ser bastante ilegibles, y además exigirían un exhaustivo proceso de prueba.

PRÁCTICAS DE AUTOEVALUACIÓN

**16.2** Use una expresión regular para verificar si un cuadro de texto contiene cualquier cantidad de dígitos.

**16.3** Una cadena debe contener la referencia a un cuadrado en un atlas. La referencia empieza con un número de página de tres dígitos, seguido por un espacio, luego por un código de letra horizontal (`A` a `F`), y finalmente un número vertical (`1` a `7`). He aquí un ejemplo:

015 D3

Escriba una expresión regular que haga una referencia a la cuadrícula.

## ● Un ejemplo de procesamiento de cadenas

---

En esta sección examinaremos la creación de un método de procesamiento de cadenas que realiza una tarea común: examinar una cadena y sustituir cada ocurrencia de una subcadena específica por otra (de longitud potencialmente distinta). Cabe mencionar que la clase `String` cuenta con un método `Replace` que también permite llevar a cabo remplazos, pero será instructivo ver el procesamiento de cadenas en acción. Para invocar el método `Replace` escribimos lo siguiente:

```
cadenaResultante = cadena1.Replace("EE.UU.", "Estados Unidos");
```

Llamaremos a nuestro método `Cambiar` para evitar cualquier conflicto con el método `Replace` existente.

He aquí un ejemplo de cómo reemplazar subcadenas. Si tenemos la cadena:

```
"ir a jugar o no ir a jugar"
```

y reemplazamos cada ocurrencia de "jugar" por "comer", crearemos la cadena:

```
"ir a comer o no ir a comer"
```

El proceso básico consiste en utilizar `IndexOf` para determinar la posición de la subcadena que nos interesa (en este caso, "jugar"). Después formamos una nueva cadena compuesta por la parte izquierda de la cadena, la parte derecha y la cadena de reemplazo al centro. Tenemos entonces lo siguiente:

```
"ir a" + "comer" + "o no ir a jugar"
```

El proceso se debe repetir hasta que no haya más ocurrencias de "jugar". Hay tres casos problemáticos:

- Cuando el usuario de `Cambiar` nos pide sustituir un valor de "", es preciso tomar en cuenta que antes de cualquier cadena puede haber un número infinito de esas cadenas vacías. Nuestra forma de proceder en este caso deberá ser devolver la cadena original sin cambios.
- Cuando la cadena de reemplazo contiene la cadena a reemplazar. Por ejemplo, podríamos tratar de cambiar "ser" por "serpiente". Para evitar que se realice un número infinito de reemplazos, nos aseguramos de que sólo se consideren subcadenas en la parte derecha de la cadena original. Para ello podríamos utilizar la variable `iniciarBúsqueda`, para guardar en dónde está el inicio de la parte derecha de la cadena en un momento dado.
- Cuando la cadena de reemplazo combinada con la cadena original produce la cadena a reemplazar. Por ejemplo, al reemplazar `bc` con `c` en `abcbcd` obtenemos `abcd`. La solución al caso anterior también contempla (y resuelve) esta posibilidad.

El código completo es:

```
private string Cambiar(string original,
                       string textoActual,
                       string textoFinal)
{
    string parteIzq, parteDer;
    int iniciarBúsqueda = 0;
    int lugar = original.IndexOf(textoActual);
```

```

if (textoActual.Length != 0)
{
    while (lugar >= iniciarBúsqueda)
    {
        parteIzq = original.Substring(0, lugar);
        parteDer = original.Substring(
            lugar + textoActual.Length,
            original.Length - lugar - textoActual.Length);

        original = parteIzq + textoFinal + parteDer;
        iniciarBúsqueda = parteIzq.Length + textoFinal.Length;
        lugar = original.IndexOf(textoActual);
    }
}
return original;
}

```

Ésta es la forma en que podríamos utilizar nuestro método:

```

string original = "ir a jugar o no ir a jugar";
string modificada = Cambiar(original, "jugar", "comer");

```

A continuación lo incorporaremos en un programa.

## ● Ejemplo práctico: Frasier

En 1970, Joseph Weizenbaum escribió un programa conocido como ELIZA, cuyo propósito era simular un estilo particular de psiquiatra. Se trataba de un programa simple, en cuanto a que se esforzaba muy poco por comprender el sentido de lo que escribían los usuarios (pacientes). Por ejemplo, si el paciente escribía:

Estoy triste

ELIZA podría responder:

usted se siente triste, ¿por qué?

De manera similar, si el paciente escribía:

Estoy C#

ELIZA podría responder:

Usted se siente C#, ¿por qué?

En esta sección presentaremos una versión aún más simplificada, a la cual llamaremos Frasier en honor al personaje de televisión estadounidense (precisamente, un psiquiatra). En esencia escribiremos un método llamado `ObtenerRespuesta`, el cual devuelve una respuesta a una pregunta que se le pasa como argumento. La cadena de entrada es introducida por el usuario mediante un cuadro de texto, y la respuesta se despliega en una etiqueta. La Figura 16.2 muestra la interfaz, y el código se presenta a continuación:

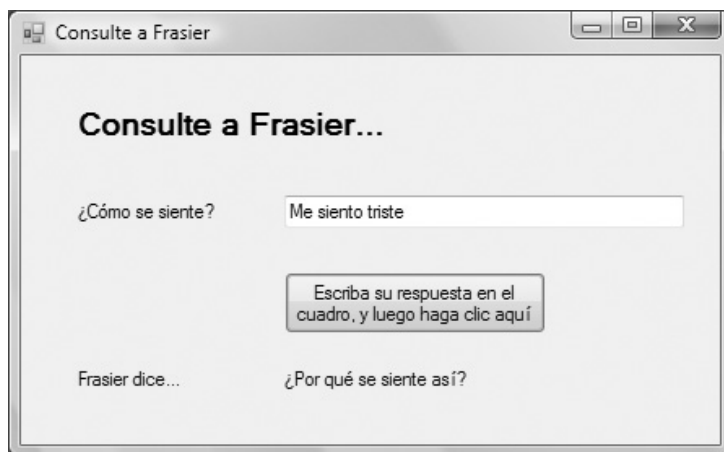


Figura 16.2 Interfaz del programa Consulte a Frasier.

```
private void button1_Click(object sender, EventArgs e)
{
    respuestaLabel.Text = obtenerRespuesta(preguntaTextBox.Text);
}
private string obtenerRespuesta(string pregunta)
{
    Random númeroAleatorio = new Random();
    int variación;
    string respuesta;
    pregunta = " " + pregunta + " ";
    variación = númeroAleatorio.Next(0, 2);
    if (variación == 0)
    {
        respuesta = transformar(pregunta);
    }
    else
        if (variación == 1)
        {
            respuesta = "¿Por qué se siente así?";
        }
        else
        {
            respuesta = "¡Por favor, sea sincero!";
        }
    return respuesta;
}
```



```

private string transformar(string pregunta)
{
    string respuestaTemp;
    if (pregunta.IndexOf(" Yo ") >= 0)
    {
        respuestaTemp = Cambiar(pregunta, " Yo ", " usted ");
        respuestaTemp = Cambiar(respuestaTemp, " me siento ", " se siente ");
        return Cambiar(respuestaTemp, " mi ", " su ") +
            "¿Por qué?";
    }
    else
    {
        if (pregunta.IndexOf(" no ") >= 0)
        {
            return "¿'No'? - ¡Qué negativo! Por favor, explíqueme.";
        }
        else
        {
            return "'" + pregunta + "' Por favor, explíqueme.";
        }
    }
}

```

Tenga en cuenta que para que este código funcione debemos añadirle nuestro método `Cambiar`. Para que las respuestas parezcan más humanas, agregamos un elemento aleatorio:

```

variación = númeroAleatorio.Next(0, 2);
if (variación == 0)
{
    respuesta = transformar(pregunta);
}
else
{
    if (variación == 1)
    {
        respuesta = "¿Por qué se siente así?";
    }
    else
    {
        respuesta = "¡Por favor, sea sincero!";
    }
}

```

El entero aleatorio provee tres casos. En dos de ellos producimos una respuesta estándar, pero en el tercero transformamos la pregunta; es decir, reemplazamos cada ocurrencia de " Yo " por " usted ". Además, añadimos espacios adicionales al principio y al final de la pregunta para ayudar a detectar palabras completas. Cabe mencionar que el programa no tiene conocimiento del significado ni de la gramática castellana. Para enriquecerlo a tal grado se requeriría un esfuerzo de programación bastante grande.

## Fundamentos de programación

- Las cadenas de caracteres contienen secuencias de caracteres.
- Existen métodos que se pueden utilizar para realizar operaciones con las cadenas de caracteres.

## Errores comunes de programación

- La clase `String` proporciona métodos y propiedades. El uso correcto de sus métodos es, por ejemplo:

```
n = cadena1.IndexOf("/");
```

en vez de:

```
n = IndexOf(cadena1);
```

- La excepción es `Length`, que es una propiedad y no usa paréntesis.
- Para declarar cadenas de caracteres en C# utilizamos la palabra clave `string`. La clase `String` nos proporciona los métodos de biblioteca.

## Secretos de codificación

Los métodos de la clase `String` requieren una cadena de caracteres sobre la cual puedan operar, como en el siguiente ejemplo:

```
string s = "demo";  
int n = s.Length;
```

Cabe mencionar que podemos proveer una cadena literal o la invocación a un método que devuelva una cadena, como en el siguiente ejemplo:

```
n = "otro demo".Length;  
n = s.Substring(0, 2).Length;    // longitud de "de"
```

## Nuevos elementos del lenguaje

El uso de `\` dentro de cadenas encerradas entre comillas para representar una sola comilla, y el uso de `@` para facilitar el uso del carácter `\` en una cadena entrecomillada.

## Nuevas características del IDE

En este capítulo no se presentaron nuevas herramientas del IDE.

## Resumen

- Las cadenas contienen una secuencia de caracteres. El primer carácter está en la posición 0.
- Para declarar y crear instancias de cadenas de caracteres podemos utilizar:

```
string s;
string nombre = "Miguel";
```

- Las herramientas más utilizadas para la manipulación de cadenas de caracteres son:

### Corrección de cadenas

```
ToUpper
ToLower
Trim
Insert
Remove
```

### Análisis de cadenas

```
Length
Substring
IndexOf
LastIndexOf
StartsWith
EndsWith
Split
CompareTo
```

### Conversión

```
Convert.ToString
Convert.ToDouble
Convert.ToInt32
Convert.ToChar
```

### Expresiones regulares

El método `IsMatch`.

Los símbolos de expresiones regulares:

```
. ^ $ [ ] { } + - \
```

## EJERCICIOS

- 16.1** Escriba un programa que reciba como entrada dos cadenas mediante cuadros de texto, y que las una. Despliegue la cadena resultante y su longitud en etiquetas.

- 16.2** Escriba un programa que reciba como entrada una cadena de caracteres y determine si es un palíndromo o no. Los palíndromos son palabras que se leen igual al derecho que al revés; por ejemplo, “oso” es un palíndromo. Suponga que la cadena no contiene espacios ni signos de puntuación.
- 16.3** Escriba un programa que reciba como entrada una cadena que puede ser un número `int` o `double` mediante un cuadro de texto. Despliegue el tipo del número. Suponga que un `double` siempre contiene un punto decimal.
- 16.4** Modifique el programa de Frier para hacerlo más humano; agregue más variación a las respuestas.

- 16.5** Escriba un programa que permita introducir datos de la siguiente forma:

```
123 + 45
6783 - 5
```

(es decir, dos enteros con el signo + o – entre ellos, y espacios que separen los distintos elementos); el programa debe mostrar el resultado del cálculo.

- 16.6** Amplíe el ejercicio 16.5 de manera que su programa reciba datos de la siguiente forma:

```
12 + 345 - 44 - 23 - 57 + 2345
```

Suponga que el usuario no cometerá errores.

(Sugerencia: el patrón de dicha entrada es un número inicial seguido de cualquier cantidad de pares operador/número. Su programa debe ser capaz de manejar el número inicial y después realizar repeticiones para trabajar con los siguientes pares.)

- 16.7** Amplíe el ejercicio 16.5, de manera que pueda recibir dos tipos de entrada:

```
setm 2 426
12 + m2
```

La instrucción `setm` debe ir seguida de dos números. El primero hace referencia a una ubicación de memoria, numerada del 0 al 9, y el segundo es un número que debe almacenarse en la memoria. Ahora se podrán realizar cálculos utilizando enteros como en los ejercicios anteriores, y también nombres de memoria, note que `m2` se refiere al dato guardado en la ubicación 2 de la memoria, (sugerencia: utilice un arreglo de enteros para representar la memoria):

```
m3 = 12 + m5 - 328 - m7
mostrar m3
```

- 16.8** Utilice una expresión regular para escribir un programa que reciba como entrada la sugerencia de una dirección de correo electrónico, e informe si es permitida o no. La dirección debe tener la forma:

```
unnombre@una.direccion
```

(Tenga en cuenta que tal vez necesite utilizar el carácter `\` como carácter de escape). La dirección no debe contener espacios ni otros signos de puntuación. Por último, amplíe el programa de manera que rechace las direcciones que contengan palabras especiales, como “webmaster”, “gobierno”, etc. El usuario deberá establecer cuáles son esas palabras especiales en un cuadro de lista.

**SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**  
.....

**16.1** Extrae todo, excepto el primero y el último caracteres, por lo que `s` se convierte en `"osibl"`.

**16.2** Después de `[0123456789]` colocamos un signo `+` para permitir la búsqueda de uno o más dígitos. También necesitamos establecer que hay dígitos al principio y al final de la cadena.

```
if (Regex.IsMatch(cadena1Box.Text, @"^[0123456789]+$"))
{
    resultadoLabel.Text="contiene dígitos";
}
else
{
    resultadoLabel.Text="no se encontraron dígitos";
}
```

**16.3** La expresión regular es:

```
^[0123456789]{3} [A-F] [1-7]$
```



# Excepciones

## En este capítulo conoceremos:

- qué es una excepción;
- por qué las excepciones son útiles;
- las herramientas de manejo de excepciones de C#.

## ● Introducción

---

En C#, el término “excepción” se utiliza para transmitir la idea de que algo ha salido mal; en términos comunes, significa que ha ocurrido un error o se ha presentado una “circunstancia excepcional”. Cabe mencionar que nos referimos a excepcional en el sentido de inusual, y no como sinónimo de maravilloso. Siendo usuario de computadoras, seguramente habrá notado que hay toda una variedad de circunstancias en las que el software puede funcionar de forma errónea, pero si es de buena calidad debe hacer frente de manera satisfactoria a los errores predecibles. Por ejemplo, las siguientes son algunas situaciones extrañas que pueden ocurrir en el manejo de un procesador de palabras, y los posibles resultados (algunas veces insatisfactorios):

- El sistema le pide que especifique el tamaño de letra como número, pero usted escribe un nombre. El sistema podría interrumpir su ejecución y regresar el control al sistema operativo, o ignorar lo que usted acaba de escribir y dejar el tamaño de letra como estaba; también es posible que despliegue un mensaje de ayuda invitándole a intentarlo de nuevo.
- Usted trata de abrir un archivo que no se encuentra en el disco. Las respuestas podrían ser similares al caso anterior.
- Usted trata de imprimir un archivo, pero su impresora se quedó sin papel. Una vez más, esta situación es predecible, por lo que el software debiera escribirse de manera que tome acciones adecuadas. Sin embargo, esto depende del hecho de que el software pueda acceder al estado actual de la impresora. En las impresoras actuales el software es capaz de examinar varios bits de estado que indican si se agotó el papel, si el dispositivo está conectado/desconectado, si hubo un atasco de papel, etc.

**PRÁCTICA DE AUTOEVALUACIÓN**

**17.1** Decida cuál sería el mejor curso de acción que el procesador de palabras podría tomar en cada uno de los casos anteriores.

Ahora veamos por qué es necesario contar con alguna forma de notificación de los errores, y cómo podríamos proporcionarla en nuestros programas.

Cuando se construyen sistemas de software y hardware, buena parte de ellos se presentan como elementos pre-empaquetados; tal es el caso de los tableros de circuitos, las clases y los métodos de C#, etc. Para simplificar el proceso de diseño es imprescindible considerar que estos elementos estarán encapsulados, de manera que, aun cuando no queremos preocuparnos por su funcionamiento interno, es vital que proporcionen alguna instrucción en caso de que ocurran situaciones de error. En consecuencia, podemos escribir el software de manera que detecte dicha situación de error y tome una acción alternativa. ¿Pero qué acción debe tomar? Ésa es la parte difícil.

Los sistemas complejos consisten en una jerarquía de métodos (es decir, métodos que invocan a otros métodos, y así sucesivamente). Algunas excepciones pueden manejarse de manera local en el método en el que ocurren, pero en casos más serios tal vez sea necesario pasar el error a otros métodos de más alto nivel. Todo depende de la naturaleza del error. En resumen, hay distintas categorías de errores que tal vez deban manejarse en distintos lugares.

He aquí una analogía que ilustra lo anterior. Imagine una organización cuyo director general da instrucciones a sus gerentes, quienes, a su vez, instruyen a los programadores y técnicos para que las lleven a cabo. Pero las cosas podrían salir mal; he aquí dos casos:

- Una de las impresoras se queda sin papel. Por lo general un técnico se hace cargo de resolver el problema pero, en el extraño caso de que la organización completa se quedara sin papel, tal vez habría que informar a uno de los gerentes.
- Un técnico se tropieza con un cable y se rompe una pierna. El director general debe hacerse cargo de las excepciones que pudieran darse en este caso (por ejemplo, una acción legal contra la empresa, etc.).

En esta analogía cada persona que hace un trabajo representa un método. El trabajo lo inició alguien de jerarquía superior a la suya. Cuando hay errores se hace indispensable contar con un plan que indique quién manejará cada tipo específico de excepción. Las herramientas de manejo de excepciones de C# nos permiten hacer esto.

Volvamos al software; como dijimos antes, las cosas pueden salir mal. ¿Pero realmente necesitamos una herramienta especial para manejar los errores? ¿No podríamos simplemente utilizar la instrucción `if`? Quizá todo se solucionaría si empleáramos un código como éste:

```
if algo sale mal
    manejar el problema
else
    manejar la situación normal
```

Aquí hemos utilizado una mezcla de castellano coloquial y sintaxis de C# para establecer el punto principal. Pero si tenemos varias invocaciones a métodos, cualquiera de las cuales podría producir un error, la lógica se vuelve tan compleja que podría viciar incluso el “caso normal”. Así, lo que habría sido una secuencia simple:

```
MétodoA()
MétodoB()
MétodoC()
```

se convertiría en:

```
MétodoA()
if MétodoA tuvo errores
    manejar el problema del MétodoA
else
    MétodoB()
    if MétodoB() tuvo errores
        manejar el problema del MétodoB
    else
        MétodoC()
        if MétodoC tuvo errores
            manejar el problema del MétodoC
```

Los casos de error (que esperamos no ocurran muy seguido) dominan la lógica, y su complejidad puede hacer que los programadores se muestren reacios a utilizar este código. De hecho, las herramientas de C# para manejar excepciones nos permiten apegarnos a la codificación para el caso normal, y manejar las excepciones en un área separada del programa.

#### PRÁCTICA DE AUTOEVALUACIÓN

**17.2** ¿De qué manera podría un método devolver un valor que indicara si funcionó o no? ¿Qué pasaría si el método devolviera un valor como parte de su tarea normal?

### ● La jerga de las excepciones

Las excepciones se inician, o se lanzan, y se detectan, o atrapan, en cualquier parte del programa. C# cuenta con las palabras clave `throw`, `try` y `catch` para llevar a cabo estas tareas. Nuestro análisis comenzará por el caso más simple: cómo atrapar una excepción iniciada por una clase de biblioteca.

### ● Un ejemplo con `try-catch`

Nuestro siguiente ejemplo es un programa simple que permite al usuario emplear un cuadro de texto para introducir un número `double` que representa uno de los lados de un cuadrado. A partir de ese dato se despliega en pantalla el área del cuadrado, como en el siguiente ejemplo:

```
El área del cuadrado mide: 6.25 unidades cuadradas
```

o un mensaje de error como éste:

```
Error: vuelva a escribir cuánto mide el lado del cuadrado
```

Veamos las nuevas características incorporadas en este programa en especial; luego examinaremos los casos generales. Las Figuras 17.1(a) y (b) muestran la ejecución de este programa con la entrada correcta, y después con la entrada incorrecta, que provoca una excepción.



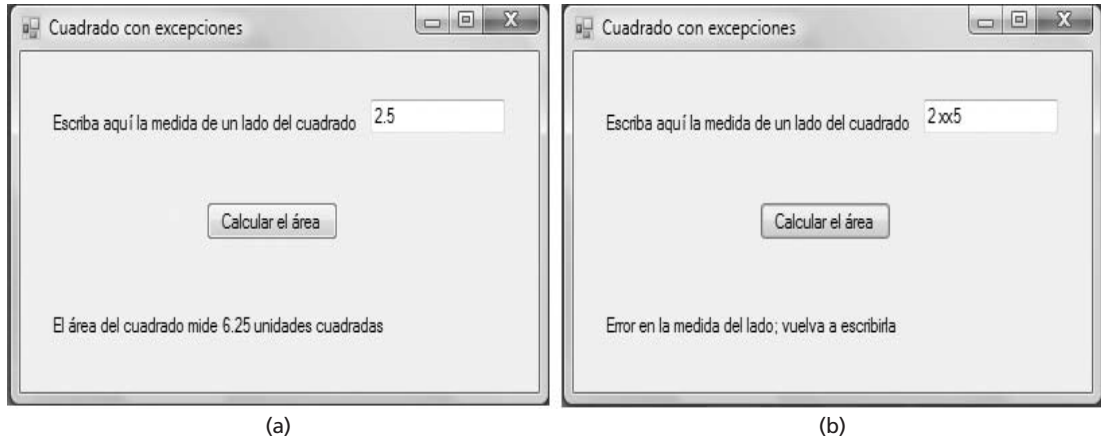


Figura 17.1 El programa Cuadrado con excepciones: (a) datos correctos; (b) se detectaron datos incorrectos.

Éste es el código:

```
private void button1_Click(object sender, EventArgs e)
{
    double lado;
    try
    {
        lado = Convert.ToDouble(textBox1.Text);
        label1.Text = "El área del cuadrado mide " +
            Convert.ToString(lado * lado) + " unidades cuadradas";
    }
    catch (FormatException objetoExcepción)
    {
        label1.Text = "Error en la medida del lado; vuelva a escribirla";
    }
}
```

Si consulta la documentación sobre el método `Convert.ToDouble`, encontrará que se lanza una excepción de la clase `FormatException`, si el argumento que enviamos es una cadena que contenga caracteres imposibles de convertir.

La siguiente es una nueva instrucción que, en esencia, actúa como una estructura de control. Su forma es:

```
try
{
    una serie de instrucciones ... (el bloque try)
}
catch (FormatException objetoExcepción)
{
    maneja la excepción de alguna forma ... (un bloque catch)
}
```

La idea es instruir a C# para que trate de ejecutar un bloque de instrucciones (el bloque “try”). Si el programa se ejecuta sin producir una excepción, las instrucciones que están después de la palabra `catch` (el bloque “catch”) son ignoradas y la ejecución continúa después del paréntesis final de la instrucción `catch`.

Sin embargo, si se produce una excepción debido a algún error en el bloque “try”, el programa deja de ejecutarse; en ese caso podemos especificar que se ejecute el bloque “catch” indicando la clase de excepción que deseamos atrapar. Para este ejemplo consultamos la documentación de la biblioteca para `Convert.ToDouble` y descubrimos que se producirá (lanzaré) una excepción de la clase `FormatException`. Si ocurre algún otro tipo de excepción no se ejecutará nuestro bloque “catch” y C# tratará de localizar en alguna otra parte de nuestro programa un bloque “catch” que especifique ese tipo de excepción. Este proceso se describe con más detalle en las secciones siguientes.

Por lo pronto examinemos cuidadosamente el bloque “catch”. En nuestro ejemplo tenemos lo siguiente:

```
catch (FormatException objetoExcepción)
{
    label1.Text = "Error en la medida del lado; vuelva a escribirla";
}
```

La línea que contiene la instrucción `catch` es algo así como la declaración de un método con un argumento. Elegimos el nombre `objetoExcepción` para el argumento. Al iniciarse la excepción `FormatException`, C# detecta que este bloque “catch” específico puede aceptar un argumento de tipo `FormatException`. C# deposita un objeto de tipo `FormatException` en `objetoExcepción`. En nuestro ejemplo no utilizamos este objeto, pero en la práctica se puede aprovechar para averiguar más sobre la excepción.

Una vez que se ejecuta el bloque “catch”, la ejecución del programa continúa con las instrucciones que están después de él. Dado que los métodos se encargan de realizar una tarea específica que no puede continuar cuando ocurre una excepción relacionada con esta tarea, es común regresar del método como en el siguiente ejemplo:

```
private void Método1()
{
    try
    {
        bloque de código
    }
    catch (UnaClaseDeExcepción objetoExcepción)
    {
        manejo de la excepción
    }
}
```

En nuestro programa de ejemplo esto es justo lo que necesitamos: el manejador de excepciones muestra un mensaje de error y da por terminado el método. Entonces el usuario puede introducir un nuevo número en el cuadro de texto.

## ● Uso del objeto de excepción

Al atrapar una excepción sabemos de antemano que algo salió mal. El objeto de excepción contiene información adicional al respecto. Para averiguarla podemos usar la propiedad `Message` y el método `ToString`, como en el siguiente ejemplo:

```
catch (FormatException objetoExcepción)
{
    MessageBox.Show(objetoExcepción.Message);
    MessageBox.Show(Convert.ToString(objetoExcepción));
}
```

- La propiedad `Message` provee una cadena de texto corta que expone un mensaje de error descriptivo. En este ejemplo el mensaje de la cadena de texto es:

La cadena de entrada no tiene el formato correcto.

- El método `ToString` devuelve una cadena con varias líneas. La primer línea es como el mensaje mencionado anteriormente, y las otras son un *rastreo de pila*, mismo que empieza en el método en donde ocurrió la excepción y recorre en sentido inverso los métodos que se invocaron y que conducen a la excepción. El rastreo termina al encontrar un bloque `catch` que coincida con la excepción. Muchas de estas líneas están relacionadas con métodos y bibliotecas del sistema de C#, pero entre ellos encontrará información relacionada con su código. El elemento clave es el número de línea de su programa que se señala. Es ahí en donde se originó directamente la excepción, o indirectamente mediante una invocación a un método. A partir de ese punto deberá comenzar a estudiar su código para descubrir el motivo de la excepción. La Figura 17.2 muestra el resultado de `ToString`, señalando la línea 320 como origen del error.

```
System.FormatException: La cadena de entrada no tiene el formato correcto.
    en System.Number.ParseDouble(String s, NumberStyles style,
NumberFormatInfo info)
    en System.Double.Parse(String s, NumberStyles style,
IFormatProvider provider)
    en System.Double.Parse(String s)
    en System.Convert.ToDouble(String value)
    en exceptioninserts.Form1.button8_Click(Object sender,
EventArgs e) en c:\mike\cslibro\cscodigoporcaps\c17exceps\
insercionesexcepciones\insercionesexcepciones.cs:línea 320
```

**Figura 17.2** Ejemplo del resultado producido por `ToString` con un objeto de excepción.

## ● Clasificación de las excepciones

En esta sección exploraremos las diversas clases de excepciones incluidas en la biblioteca de C#. Básicamente la biblioteca nos proporciona una lista de nombres de clases de excepciones, pero si no hay una adecuada podemos crear nuestra propia excepción. A pesar de esta posibilidad, aquí no abordaremos el proceso de creación de nuevas excepciones, pues confiamos en que usted podrá encontrar una que le sea útil dentro de la biblioteca.

La herencia se utiliza para clasificar los errores en tipos distintos. Hay una gran cantidad de clases de excepciones predefinidas; en la Figura 17.3 aparece una pequeña selección de las más comunes.

He aquí cómo interpretar la figura. Todas las excepciones heredan de la clase `Exception`; una de esas clases es `IOException`, la cual a su vez se divide en otras subclases, en el sentido de que `EndOfStreamException` y `FileNotFoundException` también son herederas de `IOException`. A medida que aumenta la indentación o sangría de los nombres, las excepciones se vuelven más específicas. La razón de este sistema de clasificación es que podemos optar por atrapar un solo tipo de excepción (como sucede con `EndOfStreamException`) o todo un conjunto de excepciones (como con `IOException`).

### PRÁCTICA DE AUTOEVALUACIÓN

**17.3** De acuerdo con la figura 17.3, ¿cuál es la relación entre las clases `FormatException` y `IOException`?

En la Figura 17.3 mostramos algunas excepciones comunes (y utilizaremos algunas de ellas en el capítulo 18, cuando veamos cómo trabajar con archivos). Por supuesto, éstas no son las únicas; hay

```
Exception
  SystemException
    ArithmeticException
      DivideByZeroException
      OverflowException

    CoreException
      IndexOutOfRangeException

    FormatException

    NullReferenceException

  IOException
    EndOfStreamException
    FileNotFoundException
```

Figura 17.3 Selección de clases de excepciones.

muchas más clases de excepciones. De hecho, no es suficiente conocer los nombres de las excepciones; también es necesario tomar en cuenta su origen. Siempre que vaya a utilizar un método, consulte su documentación. Ésta le indicará lo que hace, qué argumentos necesita, las excepciones que podrían iniciarse, y a qué clases pertenecen.

#### PRÁCTICA DE AUTOEVALUACIÓN

**17.4** ¿Cuáles serían las consecuencias que enfrentaría el programador si las excepciones no estuvieran clasificadas (es decir, si la estructura de excepciones sólo fuera una lista de nombres)?

### ● Bloques catch múltiples

Nuestro ejemplo del cuadrado únicamente utilizó un bloque “catch”. Sin embargo, es posible emplear varios bloques “catch” con un solo bloque `try`. Si lo hacemos, la regla establece que el bloque `catch` más específico debe ir primero, seguido de los casos más generales. En el siguiente ejemplo dividimos 100 entre un número introducido por el usuario en un cuadro de texto. Decidimos atrapar primero la excepción de división entre cero, después las excepciones de formato, y al final cualquier excepción restante. Los tres bloques “catch” deben estar en el orden siguiente:

```
private void button3_Click(object sender, EventArgs e)
{
    int inferior, superior = 100;
    try
    {
        inferior = Convert.ToInt32(textBox1.Text);
        MessageBox.Show(
            "Al dividir entre 100 obtenemos " +
            Convert.ToString(superior / inferior));
    }
    catch (DivideByZeroException objetoExcepción)
    {
        MessageBox.Show("Error: imposible dividir entre cero: vuelva a introducir
los datos");
    }
    catch (FormatException objetoExcepción)
    {
        MessageBox.Show("Error en el número: vuelva a introducirlo");
    }
    catch (SystemException objetoExcepción)
    {
        MessageBox.Show(objetoExcepción.ToString());
    }
}
```

Cabe mencionar que si invirtiéramos el orden de los últimos dos bloques “catch” nunca podría detectarse una excepción `FormatException`, ya que estaría cubierta por la clase `SystemException`.

## ● Búsqueda de un bloque catch

---

Todos nuestros ejemplos que hemos revisado muestran cómo se producen y manejan las excepciones dentro del método del clic del botón, pero podríamos elegir hacerlo en cualquier otra parte. He aquí nuestro ejemplo codificado de manera distinta:

```
private void button4_Click(object sender, EventArgs e)
{
    MessageBox.Show(
        "Este programa obtiene el cuadrado del número escrito en el cuadro de
texto");
    try
    {
        RealizarCálculo();
    }
    catch (FormatException objetoExcepción)
    {
        MessageBox.Show("Error: vuelva a introducir los datos");
    }
}

private void RealizarCálculo()
{
    double número;
    número = Convert.ToDouble(textBox1.Text);
    MessageBox.Show("el valor es " +
        Convert.ToString(número * número));
}
```

Ahora veamos el mecanismo de operación. Cuando ocurre una excepción, C# busca en el método actual (en este caso, `RealizarCálculo`) un bloque `catch` que coincida de manera exacta con la excepción, o con una superclase de la misma. Si el método en ejecución no provee un bloque que reúna tales características, C# abandona su ejecución y busca en el método que invocó al método que produjo la excepción (en este caso, el método del evento del clic de botón). En nuestro ejemplo es ahí donde encuentra un bloque `catch` apropiado y lo ejecuta. Si no encuentra un bloque `catch` que coincida, el proceso se repite y C# busca en el método que invocó al método del clic del botón. Ahora estamos en el dominio del código del sistema de C#, pero el proceso es el mismo. En algún momento se llegará al nivel más alto del código y el programa terminará, mostrando un cuadro de mensaje. Casi siempre esto no es conveniente; el objetivo de las excepciones es mantener el programa en ejecución siempre que sea posible.

Cabe mencionar que no necesitamos agrupar todos nuestros bloques `catch` en un solo método. El código siguiente muestra cómo se atrapa una excepción en `RealizarCálculo`, y las demás en el método del clic del botón.

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(
        "Este programa obtiene el cuadrado del número escrito en el cuadro de
        texto");
    try
    {
        RealizarCálculo();
    }
    catch (Exception objetoExcepción)
    {
        MessageBox.Show("La excepción se inició en RealizarCálculo()");
    }
}

private void RealizarCálculo()
{
    double número;
    try
    {
        número = Convert.ToDouble(textBox1.Text);
        MessageBox.Show("El cuadrado es " + Convert.ToString
            (número * número));
    }
    catch (FormatException objetoExcepción)
    {
        MessageBox.Show("Número incorrecto: vuelva a introducir los datos");
    }
}
```

Ésta es una descripción más general del mecanismo. Suponga que en la ejecución de cierto programa `Método1` invoca a `Método2`, el cual a su vez invoca a `Método3` (no necesitamos establecer que ésta es una ejecución específica del programa, debido a que el patrón de ejecución de los métodos podría depender de los datos que introduzca el usuario). De cualquier forma, esta ejecución produce una excepción en `Método3`. Lo que ocurre entonces es:

- Se abandona la ejecución de `Método3`.
- Si hay un bloque `catch` que coincida con la excepción en `Método3`, se utiliza y en un momento dado la ejecución del método se reanuda después del signo `}` que cierra el bloque `catch`. Tenga en cuenta que un bloque `catch` coincide con la excepción si especifica a ésta o a una superclase de la misma.

Por otra parte, si `Método3` no tiene un bloque `catch` que coincida, C# busca en los niveles jerárquico superiores, comenzando por el método que invocó a `Método3` (en este caso, `Método2`). Si encuentra ahí un bloque `catch` que coincida con la excepción, lo ejecuta y posteriormente continúa la ejecución del método, arrancando después del signo `}` que cierra el bloque `catch`.

- Si `Método2` no contiene un bloque `catch` que coincida con la excepción, C# examina el método que lo invocó (`Método1`). Este proceso de búsqueda retrospectiva en las invocaciones se denomina “propagar” la excepción. Si no se encuentra un bloque `catch` que coincida con ella, llegará el momento en que el programa se dé por terminado, y aparecerá un cuadro de mensaje con los detalles de la excepción.

## ● Lanzar una excepción: una introducción

---

En esta sección codificaremos un método que inicia (o lanza) una excepción. La tarea del método es convertir una cadena que contiene “diez”, “cien” o “mil” en su entero equivalente. Con el propósito de que este método sea útil en diversas situaciones, iniciaremos una excepción. Esto permitirá al invocador del método decidir sobre el curso de acción a tomar. En este caso optaremos por lanzar una instancia de la clase `FormatException`. He aquí el código:

```
private void button6_Click(object sender, EventArgs e)
{
    MessageBox.Show(Convert.ToString
        (PalabraANúmero("cXien")));
}

private int PalabraANúmero(string palabra)
{
    int resultado = 0;
    if (palabra == "diez")
    {
        resultado = 10;
    }

    else if (palabra == "cien")
    {
        resultado = 100;
    }

    else if ( palabra == "mil")
    {
        resultado = 1000;
    }
    else
    {
        throw (new FormatException
            ("Entrada incorrecta: PalabraANúmero"));
    }
    return resultado;
}
```

A menudo la instrucción `throw` se ejecuta como resultado de una instrucción `if`; lo que hace es crear una nueva instancia de la clase de excepción especificada, e iniciar una búsqueda tratando de encontrar un bloque `catch` que coincida. Cuando se crea una nueva excepción es posible proveer una



cadena de mensaje alternativa, misma que el método que atrapó la excepción es capaz de obtener mediante la propiedad `Message`.

## ● Posibilidades del manejo de excepciones

La metodología a utilizar en el manejo de excepciones depende de la naturaleza de las mismas y del programa en el que ocurran. En esta sección veremos dos excepciones:

- `FormatException`;
- `IndexOutOfRangeException`.

En nuestro programa del cuadrado utilizamos `FormatException`, y adoptamos el método de manejar las excepciones mostrando un mensaje de error y pidiendo al usuario que introduzca los datos de nuevo. Esto es posible (y tiene sentido) en este caso, debido a que los datos provienen directamente del usuario, quien interactúa con el programa. Pero si el error se debiera a la presencia de datos incorrectos en una base de datos (por ejemplo, una lista de calificaciones de exámenes), tal vez no sería conveniente ofrecer al usuario la oportunidad de intervenir de nuevo. En este caso lo apropiado sería mostrar un mensaje en el que le sugiriéramos ponerse en contacto con el creador de la base de datos, y dar por terminado el programa.

Ahora veamos las posibilidades que nos ofrece `IndexOutOfRangeException`. Este tipo de excepción podría provenir de un código como:

```
int [] a = new int[10];
for (int n=0; n<=10; n++)
{
    a[n] = 25;
}
```

Esta situación es diferente. Está bien atrapar la excepción, ¿pero qué hacer después? El problema no fue causado por la entrada del usuario. El ciclo `for` produce un ciclo de 0 a 10 inclusive, y por ende intenta acceder al elemento `a[10]`; el error estriba en que debimos hacer que el ciclo se ejecutara del 0 al 9. Éste es un error de programación, que en realidad deberíamos haber detectado en la etapa de prueba. Para corregirlo necesitamos realizar un proceso de depuración y volver a compilar el programa.

Entonces, ¿vale la pena atrapar los casos en los que ocurren `IndexOutOfRangeException` y `ArithmeticException`, que se podrían originar debido al mal uso de herramientas comunes como las matrices y la división? Una manera de proceder sería ignorarlas. La búsqueda de un bloque `catch` asociado a esa excepción, fallará y provocará la aparición de un mensaje en pantalla. En este caso el programa necesita depuración, y además exige la realización de más pruebas.

## ● finally

La instrucción `try` completa tiene una herramienta adicional: la instrucción `finally`. He aquí un ejemplo en una mezcla de castellano coloquial y sintaxis de C#. Suponga que la tarea consta de dos partes: la acción principal seguida de cierto código de finalización que debe ejecutarse sin importar que ocurra o no una excepción en el cálculo principal. Sin `finally` codificaríamos esto de la siguiente manera:

```
try
{
    tarea principal
    código de finalización
}
catch(FormatException objetoExcepción)
{
    mostrar un mensaje de error
    código de finalización
}
```

El uso de `finally` nos ayuda a evitar una duplicación del código de finalización, que debemos ejecutar en toda situación, como en el siguiente ejemplo:

```
try
{
    tarea principal
}
catch (FormatException objetoExcepción)
{
    mostrar un mensaje de error
}
finally
{
    código de finalización
}
```

El código que se coloca en la sección `finally` siempre se ejecutará, ocurra o no una excepción. Lo hará incluso si nuestro bloque `catch` no atrapa la excepción, o si el bloque `try` ejecuta una instrucción `return` o inicia una excepción por su cuenta.

## Fundamentos de programación

- Al escribir métodos de propósito general (tal vez desconociendo el uso que se les dará en el futuro) debemos lanzar excepciones en vez de terminar el programa u ocultar un posible error.
- Las excepciones cambian el orden en el que se obedecen las instrucciones; por lo tanto, se les considera una forma de estructura de control. Pero las excepciones no deben utilizarse para manejar casos normales. Por ejemplo, si el usuario introduce una serie de nombres y utiliza la palabra "**FIN**" para terminar la serie, la entrada "**FIN**" no es un error. En consecuencia, debe manejarse con una instrucción `while` o `if` en vez de tratarla como una excepción.

## Errores comunes de programación

- Permitir que se lance una excepción desde un método, cuando sin duda se puede manejar localmente.
- Tratar de atrapar y manejar excepciones como `ArithmeticException` y `IndexOutOfRangeException`. En pocos casos hay algo que se pueda hacer. Al atrapar estos errores podemos cometer el error de ocultarlos, cuando en realidad el programador necesita conocer su presencia.

## Secretos de codificación

- La estructura `try-catch` básica es:

```
try
{
    una serie de instrucciones
}
catch (UnaClaseDeExcepción objetoExcepción)
{
    manejo de la excepción
}
```

- Para lanzar una excepción podemos usar la siguiente instrucción:

```
throw new UnaClaseDeExcepción("mensaje de error");
```

## Nuevos elementos del lenguaje

- `try`
- `catch`
- `finally`
- `throw`
- la jerarquía de excepciones

## Nuevas características del IDE

No se abordaron nuevas herramientas del IDE en este capítulo.

## Resumen

- Las excepciones son situaciones inusuales.
- Las excepciones son instancias de clases, creadas con la palabra clave `new`.
- Los bloques `try` se utilizan para proteger código que podría lanzar una excepción.
- Los bloques `catch` pueden atrapar un tipo de excepción o una clase que contenga varias excepciones.
- El árbol de herencia de la clase `Exception` (Figura 17.2) muestra las principales excepciones con las que tenemos que lidiar, y la clase en la que se encuentran.
- Las excepciones de la clase `SystemException` son difíciles de corregir, por lo que en muchos casos podemos ignorarlas de manera intencional.

## EJERCICIOS

**17.1** Escriba un programa que proporcione dos cuadros de texto para introducir los valores enteros  $a$  y  $b$ . Muestre el resultado de  $a/b$  y de  $b/a$ . Incorpore el manejo de excepciones para los valores de cero y no numéricos que pudieran introducirse en los cuadros de texto.

**17.2** Escriba un método que resuelva ecuaciones cuadráticas. Por lo general hay dos raíces reales. La invocación al método debe tener la siguiente forma:

```
Resolver(a, b, c, raíz1, raíz2);
```

en donde las raíces se devuelven a través de parámetros de referencia. Las fórmulas para calcular las raíces son:

```
raíz1 = (-b + Math.Sqrt(b * b - 4 * a * c)) / (2 * a);
raíz2 = (b - Math.Sqrt(b * b - 4 * a * c)) / (2 * a);
```

En caso de que la discriminante  $b * b - 4 * a * c$  sea negativa, lance una excepción de tipo `ArithmeticException` con un mensaje apropiado. Escriba un método invocador que atrape su excepción.

**17.3** Si conoce la longitud de los tres lados de un triángulo, podrá calcular el área mediante la fórmula:

```
Área = Math.Sqrt(1 * (1 - a) * (1 - b) * (1 - c));
```

en donde:

```
s = (a+b+c)/2
```

Escriba un método para calcular y devolver el área. Lance una excepción apropiada (con un mensaje) cuando las tres longitudes no puedan formar un triángulo. Escriba un método invocador que atrape su excepción.

**17.4** Escriba un programa que reciba como entrada tres cadenas de caracteres a partir de tres cuadros de texto, los cuales deberán representar los números de día, mes y año. Por ejemplo, las cadenas "23", "5" y "2007" representan la fecha 23 Mayo 2007. Produzca un mensaje de error apropiado si un elemento introducido no es numérico, si no se proporciona, o si se especifica una fecha imposible. Ignore los años bisiestos.

**SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN**

**17.1** En los primeros dos casos, salir del programa no sería un buen curso de acción. Una respuesta más útil consistiría en mostrar algún tipo de señal de error y permitir que el usuario lo intente otra vez o abandone la selección del elemento. En el tercer caso la complicación es que la impresora podría quedarse sin papel a mitad de una impresión. Hay que informar al usuario sobre esto, y tal vez proporcionarle opciones para abandonar la solicitud de impresión o, suponiendo que se haya cargado más papel, seguir imprimiendo desde una página específica.

**17.2** Si el método no devolviera de antemano un valor, sería posible utilizar `return` para que regrese un valor `bool`. Ahora el método se invocaría de la siguiente manera:

```
bool error;  
error = RealizarTarea();
```

No obstante, si ya se está devolviendo un valor tendríamos que usar un parámetro por referencia (`out`), como en el siguiente ejemplo:

```
bool error;  
mayorValor = Mayor(44, 55, out error)  
if (error == true)  
    ...etc.
```

Cualquiera de estos métodos es inconveniente.

**17.3** `FormatException` hereda de `SystemException` (que a su vez hereda de `Exception`). `IOException` hereda de `Exception`. No hay relación entre las dos clases, aparte del hecho de que ambas heredan de `Exception`, al igual que todas las clases de excepciones.

**17.4** Atrapar una excepción de la que nunca se heredara sería como antes. Pero hay que considerar el caso de atrapar todas las excepciones. En ese caso necesitaríamos una enorme lista de bloques `catch`.

# 18

## Archivos

**En este capítulo conoceremos:**

- qué es un archivo de texto;
- cómo leer y escribir datos en archivos;
- cómo manipular rutas y nombres de carpetas.

### ● Introducción

---

Ya hemos utilizado archivos, puesto que empleamos el IDE para crear archivos que contienen programas de C#, y hemos usado Windows para ver una estructura jerárquica de carpetas (directorios). En este capítulo analizaremos la naturaleza de la información que podemos guardar en los archivos, y explicaremos cómo podemos escribir programas para manipularlos.

Para comenzar, aclararemos la diferencia entre la RAM (memoria de acceso aleatorio) y los dispositivos de almacenamiento de archivos (por ejemplo, discos, unidades de disco duro y CD-ROM). La capacidad de almacenamiento se mide en bytes en todos estos casos. Lo importante es dejar bien claro que existen varias diferencias importantes entre la RAM y el almacenamiento de archivos:

- La RAM se utiliza para almacenar los programas mientras que se ejecutan; antes, los archivos correspondientes son copiados de un dispositivo de almacenamiento.
- El tiempo para acceder a un elemento almacenado en la RAM es mucho menor.
- El costo de la RAM es mayor (megabyte por megabyte).
- Los datos que están en la RAM son temporales: se borran al apagar el equipo. En contraste, los dispositivos de almacenamiento de archivos pueden guardar los datos de manera permanente.

La capacidad de los dispositivos de almacenamiento de archivos es mayor que la de la RAM. Los CD-ROM tienen una capacidad de aproximadamente 740Mbytes (megabytes), y los DVD (discos versátiles digitales) pueden almacenar 4.7 Gbytes (gigabytes; cada gigabyte equivale a 1024 Mbytes). Tanto los CD-ROM como los DVD cuentan con versiones regrabables. Los discos duros comunes

pueden guardar alrededor de 80 Gbytes de información, y el modesto disco flexible cerca de 1.4 Mbytes. Pero la tecnología evoluciona con rapidez; el objetivo es crear los dispositivos de almacenamiento económicos, rápidos y pequeños que el software de computadora moderno requiere, en especial en el área de las imágenes en movimiento y del sonido de alta calidad.

## ● Fundamentos de los flujos

Los flujos nos permiten acceder a un archivo como una secuencia de elementos. El término “flujo” se utiliza en el sentido de una sucesión de datos que sale o entra del programa. A continuación explicaremos el vocabulario asociado al uso de archivos, que es similar en casi todos los lenguajes de programación. Si deseamos procesar los datos que se encuentran en un archivo existente, debemos:

1. Abrir el archivo.
2. Leer (recibir como entrada) los datos, elemento por elemento, y colocarlos en variables.
3. Cerrar el archivo cuando terminemos de trabajar con él.

Para transferir datos de variables a un archivo, debemos:

1. Abrir el archivo.
2. Enviar como salida (escribir) nuestros elementos en la secuencia requerida.
3. Cerrar el archivo cuando terminemos de trabajar con él.

Cabe mencionar que, al leer datos de un archivo, lo único que podemos hacer es trabajar elemento por elemento. Si, por ejemplo, sólo necesitamos examinar el último elemento de un archivo, tendremos que codificar un ciclo para leer un elemento a la vez hasta llegar al requerido. Para la realización de muchas tareas es conveniente visualizar los archivos de texto como una serie de líneas, cada una compuesta de varios caracteres. Cada línea se da por terminada mediante un marcador de fin de línea que puede ser en el carácter de *nueva línea*, el carácter de *retorno*, o ambos. Aun cuando el usuario se concreta a oprimir la tecla Enter para finalizar una línea, el software de Windows colocará los caracteres de nueva línea y retorno para indicar el fin de línea. C# se encarga de ocultar la mayor parte de este complicado proceso.

Además de poder manipular los archivos que contienen líneas de texto, C# también puede manejar aquellos que constan de datos binarios, como imágenes. Sin embargo, por lo general este tipo de datos se organiza en un formato más complicado dentro de los archivos.

En este capítulo utilizaremos las clases de C# que nos permiten acceder a un archivo línea por línea, como cadenas de caracteres. Esto es especialmente útil debido a que muchas aplicaciones (como los procesadores de palabras, los editores de texto y las hojas de cálculo) pueden leer y escribir en dichos archivos.

## ● Las clases `StreamReader` y `StreamWriter`

Para leer y escribir líneas de texto utilizaremos:

- El método `ReadLine` de `StreamReader`. Este método lee toda una línea de texto y la coloca en una cadena, excluyendo el marcador de fin de línea (si necesitamos dividir la línea en partes separadas podemos usar el método `split` de la clase `String`, descrito en el capítulo 16).

- La clase `StreamWriter`. Esta clase consta de dos métodos principales: `Write` y `WriteLine`. Ambos escriben una cadena en un archivo, pero `WriteLine` agrega el marcador de fin de línea después de la cadena. También podemos usar `WriteLine` sin argumentos para escribir sólo un marcador de fin de línea en el archivo. Como alternativa a `WriteLine` podemos utilizar `Write` enviándole los caracteres `\r\n` entre comillas como argumento. Más adelante veremos un ejemplo de ello.
- Los métodos `OpenText` y `CreateText` de la clase `File`. Éstos son métodos estáticos que nos proporcionan una nueva instancia de un flujo de texto. Más adelante veremos otros métodos y propiedades de la clase `File`.

Las clases de archivo forman parte del espacio de nombres `System.IO`. Como esta clase no se importa de manera automática, para hacerlo debemos usar la siguiente línea:

```
using System.IO;
```

en la parte superior de todos nuestros programas en los que se procesen archivos.

## ● Operaciones de salida con archivos

---

El programa Archivo de salida abre un archivo y escribe tres líneas en él. La interfaz de usuario sólo consiste en un botón, por lo cual no la mostraremos aquí. El código es el siguiente:

```
private void button1_Click(object sender, EventArgs e)
{
    // escribe varias líneas de texto en el archivo
    StreamWriter flujoSalida = File.CreateText(@"c:\miarchivo.txt");
    flujoSalida.WriteLine("Este archivo");
    flujoSalida.WriteLine("contiene tres");
    flujoSalida.WriteLine("líneas de texto.");
    flujoSalida.Close();
}
```

Ahora examinemos nuestro código. Primero creamos y abrimos el archivo:

```
StreamWriter flujoSalida = File.CreateText(@"c:\miarchivo.txt");
```

En este caso utilizamos el método estático `CreateText` de la clase `File`, ésta línea crea un nuevo objeto `StreamWriter` y abre el archivo. Cabe mencionar que en esta parte del código hay dos elementos que hacen referencia al archivo:

- Una cadena especificando el nombre del archivo que utiliza el sistema operativo cuando muestra las carpetas: `@c:\miarchivo.txt`. Si lo desea, puede modificar esta ruta para crear el archivo en un lugar distinto. Si omite el nombre de la carpeta y sólo utiliza `miarchivo.txt`, el archivo se creará en una carpeta llamada `debug`, que está dentro de la carpeta `bin` del proyecto de C# actual. Tenga en cuenta la necesidad de utilizar el carácter `@`, cuya función es hacer que el carácter `\` tenga su interpretación normal en vez del uso que se le da con los caracteres de escape.
- Una variable que elegimos nombrar `flujoSalida`. Es una instancia de la clase `StreamWriter` que nos proporciona el método `WriteLine`. Al utilizar `CreateText` asociamos `flujoSalida` con el archivo `"c:\miarchivo.txt"`.



Para escribir una línea de texto en el archivo utilizamos `WriteLine`, como en el siguiente ejemplo:

```
flujoSalida.WriteLine("Este archivo");
```

Si el usuario escribe los datos que deseamos colocar en el archivo (digamos, mediante un cuadro de texto), podríamos utilizar:

```
flujoSalida.WriteLine(textBox1.Text)
```

Si el archivo ya existe su contenido original será eliminado y sustituido por las tres líneas.

Por último, cerramos el archivo:

```
flujoSalida.Close();
```

Esto asegura que los datos en tránsito se escriban en el archivo, y también nos permite volver a abrirlo para lectura o escritura.

El proceso de escritura también hubiera podido llevarse a cabo mediante el uso de `Write` con los caracteres de fin de línea, como en el siguiente ejemplo:

```
flujoSalida.Write("Dos líneas\r\nde texto.\r\n");
```

Algunas veces esto es conveniente cuando manipulamos una cadena que contiene varias líneas.

En resumen, el proceso para escribir datos en un archivo consistió en:

- abrir el archivo "c:\miarchivo.txt"
- enviar (escribir) algunas cadenas al archivo;
- cerrar el archivo.

## PRÁCTICA DE AUTOEVALUACIÓN

### 18.1 Explique qué hace el siguiente código:

```
string nombreArchivo = @"c:\patron.txt";
StreamWriter flujoSalida = File.CreateText(nombreArchivo);
for (int líneas = 1; líneas <= 10; líneas++)
{
    for (int estrellas = 1; estrellas <= líneas; estrellas++)
    {
        flujoSalida.Write("*");
    }
    flujoSalida.WriteLine();
}
flujoSalida.Close();
```

## ● Operaciones de entrada con archivos

En esta sección analizaremos un programa llamado Entrada con archivo, el cual abre un archivo, recibe como entrada su contenido, y despliega éste en un cuadro de texto. La interfaz (que muestra el resultado después de oprimir el botón) aparece en la Figura 18.1. El código es el siguiente:

```
private void button1_Click(object sender, EventArgs e)
{
    //lee el archivo línea por línea
    StreamReader flujoEntrada = File.OpenText(@"c:\miarchivo.txt");
    string línea;
    línea = flujoEntrada.ReadLine();
    while (línea != null)
    {
        textBox1.AppendText(línea + "\r\n");
        línea = flujoEntrada.ReadLine();
    }
    flujoEntrada.Close();
}
```

Este código está dentro del método `button1_Click`. El archivo que elegimos como entrada fue el que creamos en el programa Salida de archivo anterior, que contiene tres líneas de texto.



Figura 18.1 Interfaz del programa Archivo de entrada.

Veamos cómo trabaja nuestro programa. Primero creamos un flujo para acceder al archivo:

```
StreamReader flujoEntrada = File.OpenText(@"c:\miarchivo.txt");
```

(Si el archivo que especificamos no es localizado, se producirá una excepción. Por ahora ignoraremos esta posibilidad, pero hablaremos sobre ella más adelante en este capítulo.)

Después utilizamos `ReadLine` para recibir como entrada la serie de líneas del archivo, y adjuntamos cada una de ellas a nuestro cuadro de texto. Hay un punto crucial aquí: no sabemos cuántas líneas tiene el archivo, por lo que establecemos un ciclo que termina cuando no haya nada más que leer:

```
línea = flujoEntrada.ReadLine();
while (línea != null)
{
    textBox1.AppendText(línea + "\r\n");
    línea = flujoEntrada.ReadLine();
}
```

Cuando `ReadLine` se queda sin líneas para leer, devuelve un objeto “vacío”. Para detectar esto podemos comparar la línea con `null`. Ésta es una palabra clave en C# para indicar que el objeto no existe.

Observe que utilizamos `ReadLine` dos veces; la que va antes de `while` se necesita para que la primera vez que el ciclo evalúe la variable `línea`, ésta tenga un valor.

El método `Append` nos permite colocar cada línea al final de cualquier texto existente en el cuadro de texto. Como `ReadLine` elimina los caracteres de fin de línea a medida que va leyendo, los volvemos a colocar en el cuadro de texto agregando los caracteres `\r\n`. Si omitiéramos este paso, el cuadro de texto contendría una línea larga sin cortes.

Dado que el archivo se lee línea por línea, podemos procesar cada una de ellas individualmente (como en el programa Búsqueda en archivos, que veremos a continuación). Otra alternativa es que, quizá sería más apropiado leer todo el archivo y colocarlo en una cadena larga, con todo y los marcadores de fin de línea. Para este fin C# nos proporciona un método llamado `ReadToEnd`, el cual utilizaremos en el programa editor de texto que analizaremos más adelante.

En resumen, el programa Archivo de entrada:

1. Abre un archivo.
2. Recibe líneas del archivo y las adjunta al cuadro de texto, mientras no llegue al fin del archivo.
3. Cierra el archivo.

## PRÁCTICAS DE AUTOEVALUACIÓN

**18.2** El siguiente código pretende desplegar la longitud de cada una de las líneas que conformen un archivo. Explique si hay algún problema con él.

```
string nombreArchivo = @"c:\temp.txt";
StreamReader flujo = File.OpenText(nombreArchivo);
string línea;
```

```

línea = flujo.ReadLine();
while (línea != null)
{
    línea = flujo.ReadLine();
    textBox1.AppendText("longitud: " + línea.Length + ",");
}
flujo.Close();

```

**18.3** Explique cualquier problema que encuentre en este código:

```

string nombreArchivo = @"c:\temp.txt";
StreamReader flujo = File.OpenText(nombreArchivo);
string línea;
línea = flujo.ReadLine();
while (línea != null)
{
    textBox1.AppendText(línea);
}
flujo.Close();

```

## ● Operaciones de búsqueda en archivos

Buscar en un archivo cierto elemento que cumpla con determinados criterios específicos es una tarea clásica. En esta sección construiremos un programa que realiza búsquedas en un archivo de calificaciones de exámenes, que tiene la siguiente forma (se omitieron acentos deliberadamente):

```

Juan Perez, 43 , 67
Dolores Gomez ,87, 99
Miguel Lopez, 54, 32
Jose Ramos, 67,43
etc...

```

Para crear este archivo podríamos escribir y ejecutar un programa de C#, o utilizar un editor de texto. Cada línea está dividida en tres áreas separadas por comas; sin embargo, permitiremos espacios adicionales. En el procesamiento de datos a dichas áreas se les conoce como campos. El programa nos permitirá escribir el nombre de un archivo, y el nombre de un estudiante, el cual supondremos que es único. Si los nombres de los alumnos se repitieran tendríamos que introducir un campo adicional para almacenar un número de identificación único para cada persona. El programa buscará en el archivo y mostrará las calificaciones del estudiante que nosotros elijamos. El código que necesitamos agregar a nuestro ejemplo anterior sobre archivos de entrada es una instrucción `while` que termine al alcanzar el final del archivo o localizar el nombre requerido. Utilizaremos el método `split` para separar los tres campos.

En vista de que el ciclo puede terminar de dos maneras, introduciremos una variable adicional llamada `encontrado` para indicar si el elemento fue localizado o no. Es de esperar que no siempre

encontremos el elemento; esto no se considera un error, por lo que lo codificamos con `if` en vez de utilizar excepciones.

Utilizando una mezcla de castellano coloquial y sintaxis de C#, la estructura de la búsqueda, sería:

```
encontrado = false
while (más líneas qué leer) And encontrado == false
{
    leer línea
    dividir línea en tres campos
    if primer campo coincide con nombre
    {
        encontrado = true
        mostrar el resto de los campos en etiquetas
    }
}
if no encontrado
{
    mostrar una advertencia
}
```

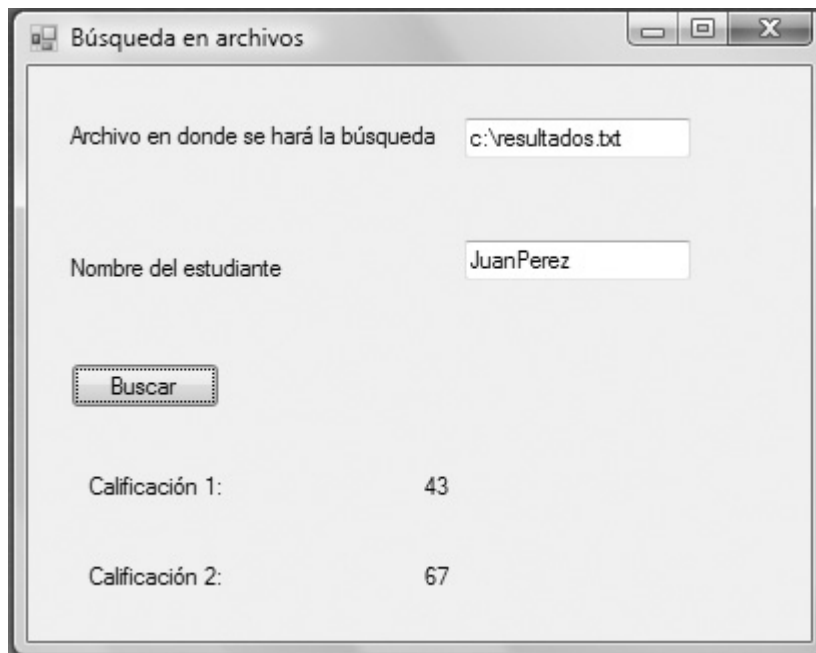


Figura 18.2 Interfaz del programa Búsqueda en archivos.

Agregamos al programa un botón de búsqueda; al hacer clic en él, el archivo (puede ser cualquiera que elija el usuario) se abre y se realiza el proceso de búsqueda. La Figura 18.2 muestra la interfaz, y el código es éste:

```
private void button1_Click(object sender, EventArgs e)
{
    string línea;
    string[] palabras = new string[3];
    char[] separador = { ',' };
    bool encontrado = false;
    StreamReader flujoEntrada;

    //borra resultados anteriores
    resultado1Label.Text = "";
    resultado2Label.Text = "";

    if (archivoTextBox.Text == "")
    {
        MessageBox.Show("Error: falta escribir el nombre del archivo");
    }
    else if (estudianteTextBox.Text == "")
    {
        MessageBox.Show("Error: falta escribir el nombre del estudiante");
    }
    else
    {
        flujoEntrada = File.OpenText(archivoTextBox.Text);
        línea = flujoEntrada.ReadLine();
        while (línea != null && encontrado == false)
        {
            palabras = línea.Split(separador);
            if (palabras[0].Trim() == estudianteTextBox.Text)
            {
                resultadoLabel1.Text = palabras[1].Trim();
                resultadoLabel2.Text = palabras[2].Trim();
                encontrado = true;
            }
            else
            {
                línea = flujoEntrada.ReadLine();
            }
        }
        if (encontrado == false)
        {
            MessageBox.Show(estudianteTextBox.Text + " no fue localizado");
        }

        flujoEntrada.Close();
    }
}
```

**PRÁCTICAS DE AUTOEVALUACIÓN**

**18.4** Modifique el programa de búsqueda de manera que ésta se lleve a cabo sin tomar en cuenta el uso de mayúsculas/minúsculas (por ejemplo, que `juan` coincida con `Juan` y con `JUAN`).

**18.5** Explique el problema que surgiría si recodificáramos nuestro ciclo de búsqueda con `||` (or), como en esta línea:

```
while (línea != null || encontrado == false)
```

## ● Archivos y excepciones

Las operaciones de entrada-salida con archivos constituyen una importante fuente de excepciones. Por ejemplo, el nombre de archivo que suministra el usuario podría ser incorrecto, el disco podría estar lleno, o el usuario podría retirar un CD-ROM del equipo mientras se está leyendo. Es cierto que podemos minimizar los efectos de introducir nombres de archivos incorrectos si utilizamos cuadros de diálogo (tema que veremos más adelante), a través de los cuales el usuario puede explorar carpetas y hacer clic en los nombres de los archivos, en vez de tener que escribirlos en cuadros de texto; sin embargo, de todas formas ocurrirán excepciones. En el capítulo 17 hablamos sobre ellas, y aquí las analizaremos por su relación con el uso de archivos.

El manejo de archivos conlleva la posibilidad de que se inicien diversas excepciones. Éstos son algunos ejemplos:

- el método `File.OpenText` puede iniciar varias excepciones; en este capítulo nos enfocaremos en la excepción `FileNotFoundException`;
- los métodos `ReadLine` (de la clase `StreamReader`) y `WriteLine` (de la clase `StreamWriter`) pueden iniciar una excepción `IOException`, junto con otras clases de excepciones.

He aquí otra versión del programa de búsqueda, incluyendo el manejo de excepciones:

```
private void button2_Click(object sender, EventArgs e)
{
    string línea;
    string[] palabras = new string[3];
    char[] separador = { ',', ' ' };
    bool encontrado = false;
    StreamReader flujoEntrada;

    //borra resultados anteriores
    resultado1Label.Text = "";
    resultado2Label.Text = "";

    if (archivoTextBox.Text == "")
    {
        MessageBox.Show("Error: falta escribir el nombre del archivo");
    }
}
```

```
else if (estudianteTextBox.Text == "")
{
    MessageBox.Show("Error: falta escribir el nombre del estudiante");
}
else
    try
    {
        flujoEntrada = File.OpenText(archivoTextBox.Text);

        línea = flujoEntrada.ReadLine();
        while (línea != null && encontrado == false)
        {
            palabras = línea.Split(separador);
            if (palabras[0].Trim() == estudianteTextBox.Text)
            {
                resultadoLabel1.Text = palabras[1].Trim();
                resultadoLabel2.Text = palabras[2].Trim();
                encontrado = true;
            }
            else
            {
                línea = flujoEntrada.ReadLine();
            }
        }
        if (encontrado == false)
        {
            MessageBox.Show(estudianteTextBox.Text
                + " no fue localizado");
        }

        flujoEntrada.Close();
    }
    catch(FileNotFoundException problema)
    {
        MessageBox.Show(problema.Message
            + ". Vuelva a introducir el nombre.");
    }

    catch(Exception problema)
    {
        MessageBox.Show("Error respecto del archivo: "
            + archivoTextBox.Text
            + ". " + problema.Message);
    }
}
```



En el ejemplo anterior consideramos `FileNotFoundException` como la excepción más probable, y produjimos un mensaje de error específico. No obstante, podrían ocurrir otras excepciones, pero éstas no son fáciles de clasificar; por lo tanto optamos por atraparlas a todas en un solo lugar, haciendo referencia a la clase `Exception`.

## ● Mensajes y cuadros de diálogo

Algunas veces necesitamos atraer la atención del usuario hacia cierta pieza de información, o pedirle que tome una decisión de vital importancia. No basta con mostrar texto en una etiqueta. C# cuenta con diversos métodos de cuadros de mensaje sobrecargados que permiten desplegar cuadros de diálogo configurados. Además, como veremos más adelante, hay cuadros de diálogo específicos para solicitar al usuario que proporcione el nombre de un archivo. Éstos son los métodos de cuadros de mensaje:

```
MessageBox.Show(mensaje);  
MessageBox.Show(mensaje, título);  
MessageBox.Show(mensaje, título, botones);  
MessageBox.Show(mensaje, título, botones, icono);
```

En cuanto a los argumentos:

- El argumento del mensaje se coloca en el centro del cuadro de mensaje.
- El argumento del título se ubica en la parte superior del cuadro de mensaje.
- El argumento de los botones es una constante que especifica los botones que necesitamos. Por ejemplo, podríamos requerir botones de sí/no. La clase `MessageBoxButtons` cuenta con los botones `AbortRetryIgnore` (cancelar/reintentar/ignorar), `OK` (aceptar), `OKCancel` (aceptar/cancelar), `RetryCancel` (reintentar/cancelar), `YesNo` (sí/no), `YesNoCancel` (sí/no/cancelar).
- El argumento de icono especifica un símbolo, como un gran signo de admiración o de interrogación. La clase `MessageBoxIcon` cuenta con los iconos `Error` (error), `Exclamation` (exclamación), `Information` (información) y `Question` (pregunta).

El método `Show` también devuelve un código que podemos examinar para averiguar en cuál botón se hizo clic. La clase `DialogResult` provee estas constantes, que podemos usar para fines de comparación: `Abort` (detener), `Cancel` (cancelar), `Ignore` (ignorar), `No`, `None` (ninguno), `OK` (aceptar), `Retry` (reintentar), `Yes` (sí). Las Figuras 18.3 y 18.4 muestran algunos ejemplos: un mensaje de advertencia y una pregunta.

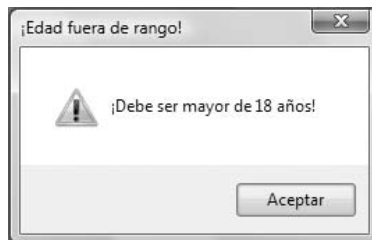


Figura 18.3 Un cuadro de mensaje de advertencia.



Figura 18.4 Cuadro de mensaje con pregunta.

He aquí el código. Observe el estilo de invocar al método de función dentro de un `if`, con lo cual se obtiene la doble función de mostrar el cuadro de mensaje y evaluar el resultado. Al ejecutar el código descubrirá que el usuario no puede regresar al formulario que desplegó el mensaje. Antes de permitirle, deberá cerrar el cuadro de mensaje de una forma u otra. A este tipo de cuadro de mensaje se le denomina *modal*.

```
//advertencia
MessageBox.Show("¡Debe ser mayor de 18 años!",
    "¡Edad fuera del rango!",
    MessageBoxButtons.OK,
    MessageBoxIcon.Exclamation);

//pregunta
if (MessageBox.Show("¿Quiere comprar este CD?",
    "Comprar CD",
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question)
    == DialogResult.Yes )
{
    MessageBox.Show("el usuario hizo clic en Sí");
}
else
{
    MessageBox.Show("el usuario hizo clic en No");
}
```

#### PRÁCTICA DE AUTOEVALUACIÓN

**18.6** Cree un código para desplegar un cuadro de mensaje que haga la siguiente pregunta:

¿París es la capital de Francia?

Despliegue respuestas adecuadas a la contestación del usuario.

## ● Cuadros de diálogo para manejo de archivos

Cuando el usuario necesita abrir un archivo desde un procesador de palabras o editor, utiliza un cuadro de diálogo que le permite explorar los directorios y hacer clic en un archivo para seleccionarlo. Por lo general este cuadro de diálogo está disponible en las aplicaciones a través del menú desplegable Archivo, que suele ubicarse en la parte superior izquierda de la ventana.

C# nos ofrece las clases `OpenFileDialog` y `SaveFileDialog` para llevar a cabo la gestión de archivos. La Figura 18.5 muestra la apariencia de un cuadro de diálogo abierto. Los pasos a seguir para programar el despliegue de cuadros de diálogo para manejo de archivos son:

- Seleccionamos la opción `OpenFileDialog` en el cuadro de herramientas, y arrastramos el objeto hasta el formulario. El cuadro de diálogo se desplazará a la bandeja de componentes, ya que no reside en el formulario. El primer cuadro de diálogo que creemos se llamará `openFileDialog1`.
- A continuación establecemos sus propiedades según se requiera; podemos hacerlo en tiempo de diseño, si es que las conocemos. Una de las propiedades más útiles de este objeto es `InitialDirectory`, una cadena que contiene la ruta del directorio inicial.
- Para desplegar el cuadro de diálogo utilizamos su método `ShowDialog`, el cual también devuelve un resultado al programa. Este resultado indica la manera en que el usuario cerró el cuadro de

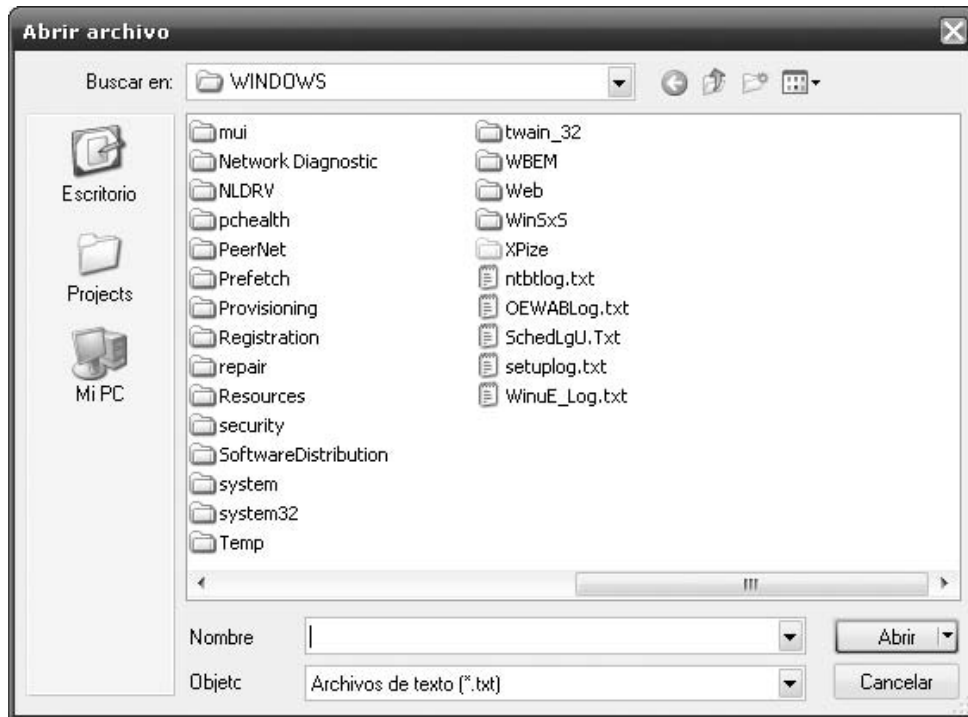


Figura 18.5 Un cuadro de diálogo `OpenFileDialog` en acción.

diálogo; por ejemplo, haciendo clic en el botón **Abrir** o **Cancelar**. La clase `DialogResult` contiene una lista de constantes con nombres convenientes que podemos utilizar.

- Empleamos también la propiedad `FileName`, la cual nos provee una cadena que contiene el nombre del archivo seleccionado.

He aquí un ejemplo de código de C# para crear un cuadro de diálogo `OpenFileDialog`:

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
    MessageBox.Show(openFileDialog1.FileName);
}
```

En este caso utilizamos `ShowDialog` en una instrucción `if` que compara el número devuelto por `ShowDialog` con la constante `DialogResult.OK` (es menos probable cometer errores si usamos las constantes proporcionadas por C# que los números por sí solos). En este contexto, “OK” significa que se hizo clic en el botón **Abrir**. Si el usuario cerró el cuadro de diálogo usando el botón “X” de la esquina superior derecha de la ventana, o canceló el cuadro de diálogo, no realizamos acción alguna. El proceso para construir y usar un cuadro de diálogo `SaveFileDialog` es casi idéntico. El programa editor de texto que analizaremos a continuación muestra estos cuadros de diálogos, y un menú que contiene las opciones **Guardar**, **Abrir** y **Guardar como**.

Cuando haya comprendido la operación de los cuadros de diálogo le recomendamos utilizarlos en vez de usar cuadros de texto para recibir el nombre de un archivo. Son un poco más difíciles de programar, pero mucho más resistentes a los errores que pudieran cometerse al escribir los nombres de archivos.

## ● Creación de menús

Muchas aplicaciones proveen un amplio rango de herramientas, con la consecuencia de que resulta impráctico asignar un botón para iniciar cada una de esas herramientas: de hacerlo, se consumiría demasiado del valioso espacio en pantalla. El menú es una solución a ese problema. Ocupa muy poco espacio y sus opciones sólo son visibles cuando se hace clic en él.

En C# podemos crear menús que se desplegarán en la parte superior de un formulario; para ello seleccionamos el control `MenuStrip` del cuadro de herramientas, y lo colocamos en el formulario (a continuación C# abre la bandeja del sistema y coloca el menú ahí). Al principio C# nos pregunta si deseamos que el menú sea `MenuItem` (menú de elementos/opciones), `ComboBox` (cuadro combinado) o `TextBox` (cuadro de texto). En este caso seleccione la opción `MenuItem`, tras lo cual aparecerán algunos indicadores “Escriba aquí”, como se ilustra en la Figura 18.6. Estos indicadores nos permiten establecer el texto de las opciones de que constará el menú; en este momento, podemos:

- crear un nuevo elemento del menú, escribiendo debajo del menú existente, o
- crear el encabezado de una nueva lista de elementos de menú, escribiendo a la derecha del nombre de menú existente.

En este caso creamos un menú llamado “Archivo” y le agregamos los elementos **Abrir...**, **Guardar**, **Guardar como...** y **Salir**. En Windows la convención es colocar el menú Archivo a la izquierda del formulario, y utilizar “...” para indicar que aparecerán más opciones. Al crear un elemento de menú podemos cambiar su nombre en tiempo de diseño haciendo clic en él con el botón derecho del ratón y seleccionando la opción (**Name**) en la ventana Propiedades. El siguiente es el código para crear un programa llamado Editor de texto; su interfaz se muestra en la Figura 18.7.

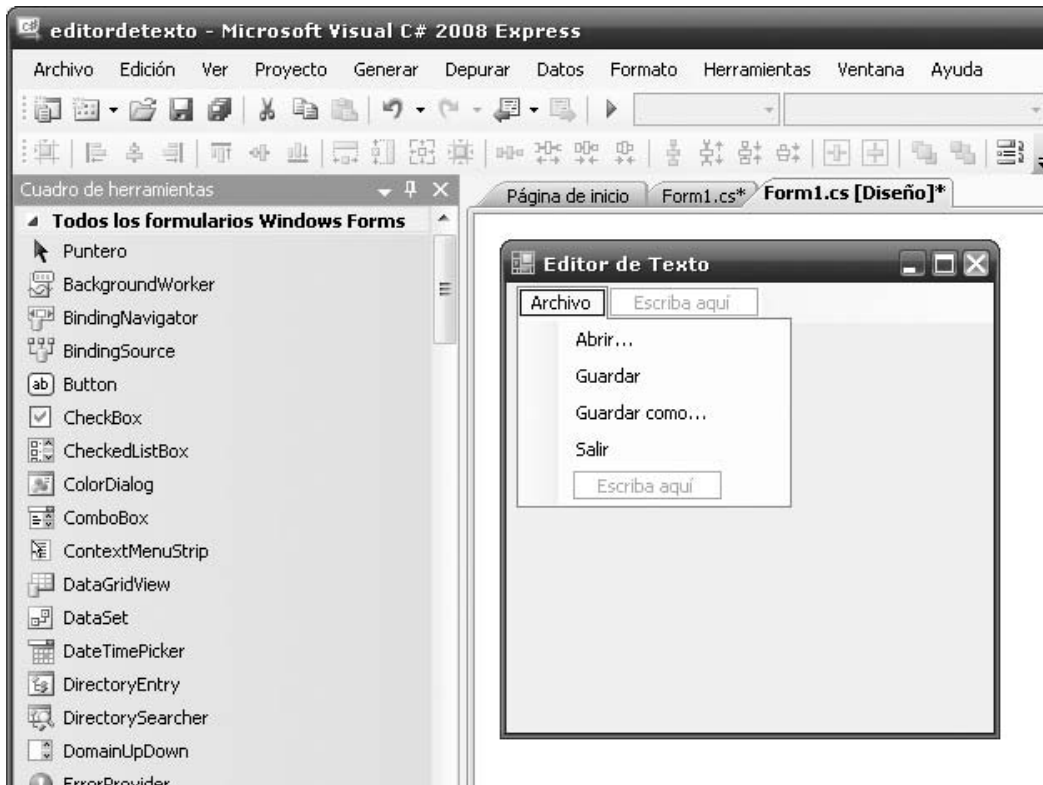


Figura 18.6 Creación de un menú en tiempo de diseño.

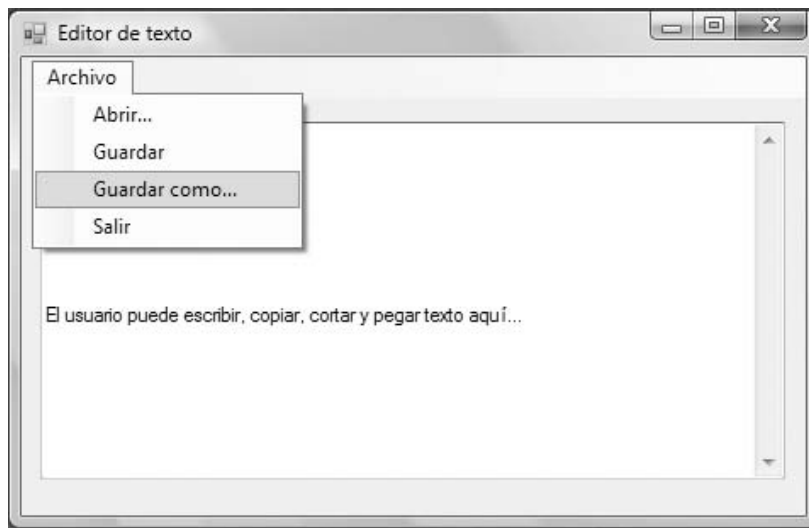


Figura 18.7 Interfaz del programa Editor de texto.

Como puede ver, cambiamos el nombre de los elementos del menú por `menúAbrir`, `menúGuardar`, `menúGuardarComo` y `menúSalir`. En consecuencia, C# crea los métodos de manejo de eventos basándose en esos nombres.

```
private string archivoActual = ""; // variable de instancia

private void menúAbrir_Click(object sender, EventArgs e)
{
    StreamReader flujoEntrada;
    openFileDialog1.InitialDirectory = @"c:\";
    if (openFileDialog1.ShowDialog() == DialogResult.OK)
    {
        archivoActual = openFileDialog1.FileName;
        flujoEntrada = File.OpenText(archivoActual);
        textBox1.Text = flujoEntrada.ReadToEnd();
        flujoEntrada.Close();
    }
}

private void menúGuardar_Click(object sender, EventArgs e)
{
    if (archivoActual != "")
    {
        StreamWriter flujoSalida = File.CreateText(archivoActual);
        flujoSalida.Write(textBox1.Text);
        flujoSalida.Close();
    }
}

private void menúGuardarComo_Click(object sender, EventArgs e)
{
    StreamWriter flujoSalida;
    saveFileDialog1.InitialDirectory = @"c:\";
    if (saveFileDialog1.ShowDialog() == DialogResult.OK)
    {
        archivoActual = saveFileDialog1.FileName;
        flujoSalida = File.CreateText(archivoActual);
        flujoSalida.Write(textBox1.Text);
        flujoSalida.Close();
    }
}

private void menúSalir_Click(object sender, EventArgs e)
{
    Application.Exit(); // termina inmediatamente
}
```

El uso de menús es bastante simple, ya que C# proporciona los encabezados de los métodos de eventos; todo lo que necesitamos hacer es agregar un cuadro de texto y el acceso a los archivos. Éstas son algunas consideraciones respecto del programa:

- Casi toda el área del formulario es utilizada por el cuadro de texto, como podemos ver en la Figura 18.7.
- Las propiedades del cuadro de texto se configuraron de manera que tenga una barra de desplazamiento vertical; además, su propiedad `Multiline` se especificó como `true` de manera que admita la introducción de varias líneas.
- Cuando el usuario solicita abrir un archivo, el cuadro de diálogo está configurado para iniciar la búsqueda en la carpeta `c:\`.
- Para leer el archivo utilizamos el método `ReadToEnd` de la clase `StreamReader`. Este método lee todo el archivo (desde la posición actual) y lo coloca en una cadena de caracteres, la cual contendrá marcadores de fin de línea al igual que el texto visible. Para almacenar esta cadena completa en el cuadro de texto usamos la siguiente línea:

```
textBox1.Text = flujoEntrada.ReadToEnd();
```

- De manera similar, podemos escribir con una sola instrucción todo el cuadro de texto en un archivo:

```
flujoSalida.Write(textBox1.Text);
```

- La variable `archivoActual` se declara fuera de los métodos, como una variable de instancia, ya que varios métodos la utilizan.
- Para tener consistencia con las demás aplicaciones de Windows, agregamos la opción `Salir` a nuestro menú. El método estático `Exit` de la clase `Application` hace que el programa termine de inmediato.

El programa Editor de texto pone en evidencia el poder de los componentes y el entorno de programación de C#:

- para crear el menú sólo tuvimos que introducir las opciones y modificar sus nombres;
- los cuadros de diálogo para manejo de archivos proveen un acceso familiar a dichos elementos;
- el cuadro de texto multilinea permite editar el texto. El botón derecho del ratón también provee acceso a las herramientas cortar y pegar del portapapeles.

## PRÁCTICA DE AUTOEVALUACIÓN

**18.7** En el programa Editor de texto, al guardar los cambios se sobrescribe la versión anterior del archivo. Así es como trabajan casi todos los editores y procesadores de palabras. Modifique este comportamiento; proporcione un cuadro de diálogo que pregunte al usuario si realmente desea guardar (es decir, sobrescribir) el archivo.

## ● La clase Directory

Esta clase contiene herramientas para manipular archivos y directorios (carpetas) completos. Estas características no están relacionadas con el acceso a los datos dentro de los archivos. Usted puede utilizar la clase `Directory` sin necesidad de usar flujos de entrada/salida cuando desee manipular los nombres de los archivos en vez de su contenido. La Figura 18.8 muestra la interfaz de un programa que despliega directorios y archivos dentro de un directorio seleccionado; analizaremos su código a continuación.

Antes de seguir es necesario que hablemos sobre terminología. Como sabe, los sistemas operativos proporcionan una estructura jerárquica, estableciendo las rutas a los archivos de la siguiente forma:

```
c:\datos\exámenes.txt
```

Ésta es una ruta estilo Windows, en donde se utiliza el carácter `\` como separador. De acuerdo con la terminología usual:

- el elemento completo es una ruta;
- la extensión del archivo es `txt`;
- la ruta del directorio es `c:\datos`;
- el nombre del archivo dentro de esa ruta es `exámenes.txt`;
- si un programa hace referencia a un archivo sin proveer el nombre de un directorio (por ejemplo, si sólo utiliza `exámenes.txt`), se asume que el archivo se encuentra en el mismo directorio en que está almacenado el programa, y desde el cual se ejecuta.

Enseguida le proporcionaremos detalles sobre algunos métodos seleccionados de la clase `Directory`. Estos métodos son estáticos y se utilizan para operar sobre cadenas que contienen rutas de archivos.

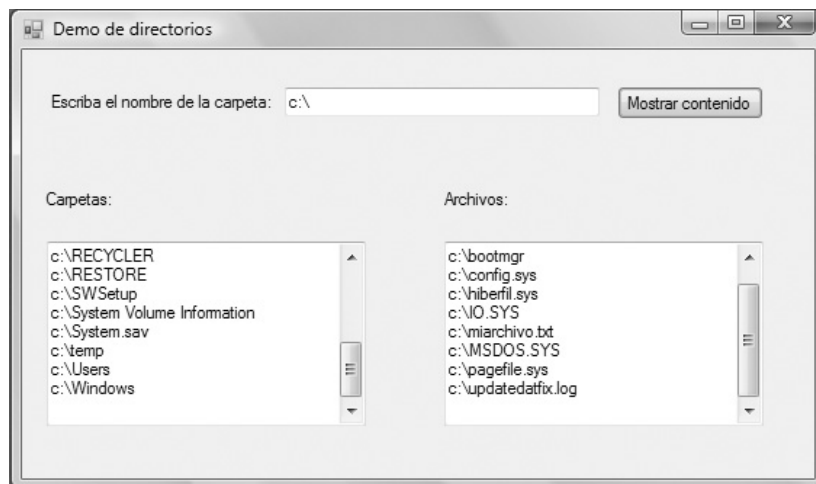


Figura 18.8 Interfaz del programa Demo de directorios.



### GetFiles

Este método opera sobre una cadena que contiene la ruta de un directorio. Devuelve un arreglo de cadenas que incluyen todos los nombres de los archivos que están en el directorio especificado. El siguiente programa (Demo de directorios) muestra este método en acción.

Algunas veces puede ser necesario comprobar de qué tipo es un archivo antes de abrirlo. He aquí un ejemplo que comprueba si el nombre de un archivo termina en `.txt`. En Windows también pueden utilizarse mayúsculas, por lo que debemos convertir el nombre de archivo a mayúsculas antes de revisarlo. En este caso podemos usar el método `EndsWith`:

```
string nombreArchivo = @"c:\pruebas\demo.txt";
if (nombreArchivo.ToUpper().EndsWith(".TXT"))
{
    MessageBox.Show("se trata de un archivo de texto");
}
```

### GetDirectories

Este método opera sobre una cadena que contiene el nombre de un directorio. Devuelve un arreglo de cadenas que incluyen los nombres de todos los directorios que están en el directorio especificado.

```
string []dirs = Directory.GetDirectories(@"c:\");
```

El programa Demo de directorios muestra este método en acción. Cabe mencionar que este método no recorre todo un árbol de directorios hasta el final; sólo avanza hasta un nivel de profundidad. Es necesario invocar varias veces `GetDirectories` para progresar a niveles más profundos.

El programa Demo de directorios permite que el usuario introduzca el nombre de un directorio; en respuesta, se desplegará el nombre de todos los archivos y directorios que se encuentren en el directorio elegido.

Cabe mencionar que la clase `Directory` se encuentra dentro del espacio de nombres `System.IO` por lo que es necesario incluir la línea “using `System.IO`;” al inicio del programa.

```
private void button1_Click(object sender, EventArgs e)
{
    //muestra todos los nombres de archivos
    string[] archivos = Directory.GetFiles(nombreCarpetaTextBox.Text);
    for (int conteo = 0; conteo < archivos.Length; conteo++)
    {
        archivosTextBox.AppendText(archivos[conteo] + "\r\n");
    }

    //muestra todos los nombres de directorios (carpetas)
    string[] dirs = Directory.GetDirectories(nombreCarpetaTextBox.Text);
    for (int conteo = 0; conteo < dirs.Length; conteo++)
    {
        carpetasTextBox.AppendText(dirs[conteo] + "\r\n");
    }
}
```

Una vez que hayamos llenado nuestros arreglos, podremos usar su propiedad `Length` para controlar un ciclo que analice cada elemento, de uno en uno.

## Fundamentos de programación

- Los programas usan flujos para leer datos de los archivos y escribir datos en ellos.
- Utilizamos archivos para preservar los datos después de ejecutar un programa, o para pasar datos a otros programas.

## Errores comunes de programación

- Olvidar colocar:  

```
using System.IO;
```

en la parte superior de los programas que procesen archivos.
- Olvidar colocar @ antes de cualquier cadena entre comillas que contenga rutas de archivos.

## Secretos de codificación

- Para declarar y crear flujos de archivos podemos usar el siguiente código:

```
StreamReader flujoEntrada = File.OpenText(@"c:\miarchivo.txt");  
StreamWriter flujoSalida = File.CreateText(@"c:\miarchivo.txt");
```

- Para leer un archivo línea por línea usamos este código:

```
string línea;  
línea = flujoEntrada.ReadLine();  
while (línea != null)  
{  
    // procesar la línea...  
    línea = flujoEntrada.ReadLine();  
}
```

o leemos todo el archivo a la vez mediante el método `ReadToEnd`.

- Para cerrar flujos podemos usar esta línea de código:

```
flujoSalida.Close();
```

## Nuevos elementos del lenguaje

En este capítulo introducimos las siguientes clases:

- `File`
- `Directory`
- `StreamReader` y `StreamWriter`
- `FileNotFoundException`
- `SaveFileDialog` y `OpenFileDialog`

## Nuevas características del IDE

- Podemos colocar un menú principal en un formulario, y manejar los clics en sus elementos mediante un método generado por C#.
- Es posible colocar cuadros de diálogos de archivos en un formulario (mismos que se almacenarán en la bandeja de componentes). Además, podemos manipular sus propiedades en tiempo de diseño y tiempo de ejecución.

## Resumen

- Los archivos se abren, después se escribe en ellos (o se lee de ellos), y por último se cierran.
- La clase `Directory` nos permite acceder al contenido de un directorio. Sus métodos nos permiten acceder a los nombres de sus subdirectorios y sus archivos.

## EJERCICIOS

- 18.1** Escriba un programa que coloque su nombre y dirección en un archivo llamado `direccion.txt`. Use un editor para comprobar que el archivo tenga el contenido esperado.
- 18.2** Escriba un programa para contar el número de líneas de que consta un archivo. Asegúrese de que funcione también en archivos vacíos (en cuyo caso deberá producir como resultado un valor de cero).
- 18.3** Escriba un programa que examine cada línea de un archivo para ver si contiene una cadena específica. Adjunte cada una de las líneas encontradas a un cuadro de texto multilíneas, anteponiendo el nombre del archivo. Use otro cuadro de texto para guardar la cadena a buscar.
- 18.4** Escriba un programa que lea dos archivos línea por línea y compare si son iguales. Además, deberá desplegar un mensaje que indique si los archivos son iguales o no. Para obtener los nombres de los archivos utilice dos cuadros de diálogo para manejo de archivos.
- 18.5** Escriba un programa que reemplace una cadena por otra en cada línea de un archivo, y que escriba la nueva versión en otro archivo. Use dos cuadros de texto para las cadenas “de” y “para”. Puede usar el método `cambiar` que creamos en el capítulo 16.
- 18.6** Este ejercicio implica escribir dos programas que ayuden a extraer partes de archivos y las inserten en otros. Podría utilizarlos, por ejemplo, para insertar muestras de código en un ensayo, o para colocar encabezados estándar al inicio de un archivo.
  - (a) Escriba un programa que lea un archivo línea por línea; y copie las líneas entre los marcadores especiales a un archivo de salida. He aquí el ejemplo del contenido de un archivo de entrada:
 

```
Aunque casi todos los guitarristas de blues
son diestros (o guitarristas zurdos que
cambiaron el encordado de sus guitarras,
como Jimi Hendrix), dos guitarristas zurdos
han dejado huella en ese género musical:
EXTRAERA:king.txt
```

```

Albert King, cuyo estilo se basa en
acordes con estiramientos extensos que se tocan lentamente.
FINEXTRAER
EXTRAERA: rush.txt
Otis Rush, un escritor y cantante
muy popular.
FINEXTRAER
Sin embargo, dicha metodología restringe
al guitarrista a utilizar acordes simples.

```

Las líneas entre el primer extracto deben copiarse al archivo `king.txt`, y las que están entre el segundo extracto deben copiarse al archivo `rush.txt`.

- (b) Escriba un programa que copie un archivo línea por línea a otro archivo. Cualquier instrucción "insertar" hará que el archivo especificado se inserte en el archivo de salida. He aquí un ejemplo:

```

Lista de guitarristas:
INSERTARDE:king.txt
INSERTARDE:rush.txt

```

Suponga que los archivos son como los anteriores, sin errores en el uso de las líneas especiales para insertar/extraer.

- 18.7** Escriba un programa que calcule el número total de líneas almacenadas en todos los archivos de texto que hay en una carpeta específica.
- 18.8** Escriba un programa que examine todos los archivos `txt` que estén en una carpeta seleccionada en busca de líneas que contengan una cadena específica. Dichas líneas deberán adjuntarse a un cuadro de texto multilíneas, anteponiendo el nombre del archivo al que correspondan. Use un cuadro de texto que permita al usuario introducir la cadena a buscar.
- 18.9** Escriba un programa que adjunte las primeras diez líneas de cada archivo `txt` que hay en una carpeta seleccionada y las coloque en un cuadro de texto multilíneas, anteponiendo el nombre del archivo al que correspondan. Si el archivo contiene menos de diez líneas, el programa deberá mostrar todo su contenido.

## SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

- 18.1** Crea un archivo que contiene el siguiente texto:

```

*
**
***
etc

```

Si omitimos el uso de `writeLine`, todas las estrellas aparecerían en una sola línea larga. En vez de `writeLine` podríamos usar:

```

flujoSalida.Write("\r\n");

```

**18.2** Cuando se devuelve `null` al final del archivo, ocurre un error "NullReference Exception". Necesitamos invertir el orden de las dos instrucciones dentro del ciclo para que inmediatamente después de usar `ReadLine` el programa regrese al inicio del ciclo y se evalúe la condición de final de archivo. Además, perdemos el texto de la primera línea.

**18.3** El programa itera indefinidamente (o por lo menos hasta que el cuadro de texto se llena), ya que no hay `ReadLine` dentro del ciclo.

**18.4** Podemos convertir cada elemento a mayúsculas, por ejemplo:

```
if (palabras[0].ToUpper() == estudianteTextBox.Text.ToUpper())
```

**18.5** Los ciclos `while` se repiten hasta que la condición completa se evalúe como falsa. Suponga que tenemos dos condiciones que al principio son verdaderas, y hay con un operador `||` entre ellas. Que una de las partes se evalúe como falsa no provoca que toda la condición se haga falsa, ya que la otra condición seguirá siendo verdadera.

Por otro lado, si conectamos las dos condiciones con `&&` (y), si una condición cambia a falsa toda la condición será evaluada como falsa. Es correcto usar `&&` en nuestra búsqueda. En nuestro ejemplo incorrecto con `||` ocurre lo siguiente: si los datos no se encuentran no hay forma de salir del ciclo, ya que nunca podrán ser falsas ambas condiciones. El programa iterará indefinidamente.

**18.6** Debemos proporcionar un icono de interrogación y botones sí/no, como en el siguiente ejemplo:

```
if (MessageBox.Show("¿París es la capital de Francia?",
    "Pregunta",
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question)
    == DialogResult.Yes )
{
    MessageBox.Show("Sí, ;su respuesta es correcta!");
}
else
{
    MessageBox.Show("Lo siento... Su respuesta es incorrecta.");
}
```

**18.7** Hay que agregar el siguiente código al evento del menú Guardar:

```
if (MessageBox.Show("¿Realmente desea guardar el archivo?",
    "Editor de Texto",
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question)
    == DialogResult.Yes )
{
    MessageBox.Show("el usuario hizo clic en Sí");
    //código para guardar el cuadro de texto en el archivo ...
}
```

# Programas de consola

## En este capítulo conoceremos cómo:

- crear una aplicación de consola;
- usar las operaciones de entrada y salida con la consola;
- procesar los argumentos de la línea de comandos;
- ejecutar programas desde el símbolo del sistema;
- usar secuencias de comandos con archivos de procesamiento por lotes.

## ● Introducción

---

La interfaz de usuario ha cambiado mucho en cuanto a capacidades (y complejidad) desde los primeros días de la computación. En la década de los sesenta era común interactuar con un dispositivo conocido como teletipo (o consola de operación), el cual en esencia era una máquina de escribir eléctrica con un rollo de papel para mostrar texto. Todo lo que el usuario podía hacer era escribir una línea de texto que el programa interpretaría como un comando o como datos.

En estos tiempos de pantallas de alta resolución, ratones, tapetes de tipo táctil y capacidad de introducción de datos mediante voz, tal vez le parezca extraño que un estilo de interacción mediante líneas de texto pueda ser útil, pero en realidad lo es. Algunos tipos de programa no requieren interacción por parte del usuario. He aquí algunos ejemplos:

- Un programa que calcula la cantidad de espacio libre en el disco duro. Todo lo que el usuario necesita hacer es indicar cuál dispositivo debe examinarse; el programa también podría ejecutarse de manera automática, tal vez al eliminar archivos.
- Un programa que imprime la lista de empleados de una empresa. Dado que estas listas no cambian muy seguido, todo lo que necesitamos hacer es ejecutar el programa.

- Un programa que forme parte de una tarea más grande y que se ejecute sin intervención humana. A dicha metodología se le conoce como “secuencia de comandos” o programación “por lotes”. Por ejemplo, cuando usted inicia un sistema operativo, se ejecuta una secuencia de comandos (un archivo que contiene instrucciones para ejecutar programas).

## ● Nuestro primer programa de consola

A continuación crearemos un pequeño programa que pide su nombre al usuario y después lo despliega en pantalla. Los pasos para crear un proyecto de consola son casi idénticos al proceso para crear un programa de GUI, sólo que debemos seleccionar “Aplicación de consola” como tipo de proyecto. Después asigne un nombre a su proyecto, en este caso `Hola`. El IDE mostrará código que contiene un método llamado `Main`, como en el siguiente ejemplo:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Hola
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Ahora debemos modificar el método `Main` para que quede así:

```
static void Main(string[] args)
{
    string nombre;
    Console.WriteLine("Escriba su nombre:");
    nombre = Console.ReadLine();
    Console.WriteLine("Hola, " + nombre);
    string espera = Console.ReadLine();
}
```

Por último, utilice el comando **Guardar como** y escriba el nombre `Hola.cs`. Ejecute el programa, escriba su nombre y presione la tecla **Enter**. Observe el resultado y oprima de nuevo **Enter** para terminar el programa. La Figura 19.1 muestra el programa en ejecución.

Ahora veamos cómo funciona el programa:

- Cuando el programa se ejecuta se hace una invocación automática al método llamado `Main`; este método es el que da comienzo a todo.



Figura 19.1 Ejecución de la aplicación de consola `Hola`.

- El método `Main` no forma parte de un objeto creado con la palabra clave `new`, por lo tanto, se declara como `static`.
- La clase `Console` está contenida en el espacio de nombres `System` y cuenta con métodos similares a los que utilizamos para acceder a los archivos. Tenemos:
  - `WriteLine`, que escribe una línea en la pantalla, seguida de un carácter de fin de línea. (El método `Write` es similar, sólo que no agrega el fin de línea.)
  - `ReadLine`, que recibe del teclado una línea de texto completa, y la interpreta como cadena de texto. El marcador de fin de línea no se almacena en esta cadena.
- Cuando las aplicaciones de consola terminan su ejecución, la ventana se cierra automáticamente. Como veremos más adelante, esto es lo que se requiere en muchos casos, pero para nuestro primer programa es inconveniente, debido a que no nos permite ver la salida. Por ello, para evitar que se cierre la ventana insertamos una línea adicional:

```
string espera = Console.ReadLine();
```

Colocamos esta línea justo antes del paréntesis de cierre del método `Main`. Su propósito es que el programa espere a que el usuario escriba otra línea de texto, o simplemente que presione la tecla **Enter**. Le recomendamos usar este código en las primeras etapas de creación y depuración de sus aplicaciones de consola, para eliminarla cuando la versión final esté lista.

- Este proyecto específico se llama `Hola`, y al archivo de C# también le pusimos ese nombre. El nombre del proyecto no necesita ser el mismo que el nombre del archivo de C#, pero en este caso era apropiado que lo fuera. El IDE crea la versión ejecutable en el archivo:

```
\\Hola\\bin\\Debug\\Hola.exe
```

El nombre `bin` es abreviatura de “binario”.

El nombre del archivo ejecutable, `Hola.exe`, se deriva del nombre del archivo de C#.



## ● El símbolo del sistema: `cd` y `dir`

Seguramente usted está familiarizado con el proceso de hacer clic en los iconos de las carpetas para navegar por su contenido, pero al ejecutar aplicaciones de consola algunas veces es necesario utilizar comandos de teclado para realizar las mismas tareas. En este caso las instrucciones se introducen en el *símbolo del sistema*, el cual puede ejecutarse desde el menú **Inicio** siguiendo esta ruta:

**Inicio | Todos los programas | Accesorios | Símbolo del sistema**

(Cabe mencionar que los detalles de la ruta varían ligeramente dependiendo de la versión de Windows que utilice. En algunas de ellas no es necesario navegar hasta el menú **Accesorios**, y en otras esta característica recibe el nombre de MS-DOS.)

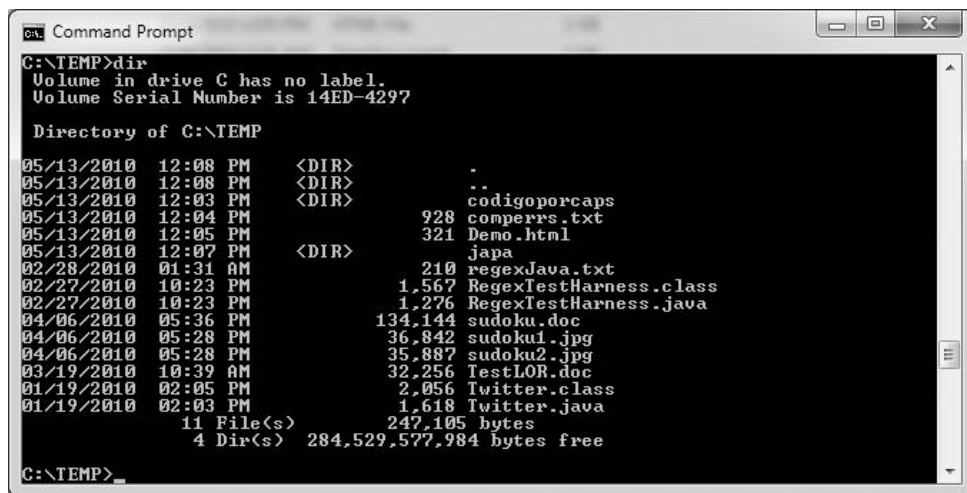
Cuando el símbolo del sistema se ejecuta aparece una ventana de color negro en la que podemos escribir comandos. Para navegar entre los directorios utilizamos los comandos `dir` (directorio) y `cd` (cambiar directorio). Estos comandos ponen a su disposición muchas opciones, pero aquí le presentaremos las más útiles; además, hay que tener en cuenta que emplean una terminología antigua, en la cual a las “carpetas” se les denominaba “directorios”. No obstante, ambos términos se refieren a lo mismo.

### El comando `dir`

Al ejecutar el símbolo del sistema aparece una línea como ésta:

```
C:\TEMP>
```

Esta línea nos indica que estamos en la unidad `C:`, en el directorio `TEMP`. Al escribir `dir` aparecen los nombres de todos los directorios y archivos almacenados en el directorio actual, como puede verse



```
C:\TEMP>dir
Volume in drive C has no label.
Volume Serial Number is 14ED-4297

Directory of C:\TEMP

05/13/2010  12:08 PM  <DIR>          .
05/13/2010  12:08 PM  <DIR>          ..
05/13/2010  12:03 PM  <DIR>          codigoporcaps
05/13/2010  12:04 PM                928 comperrs.txt
05/13/2010  12:05 PM                321 Demo.html
05/13/2010  12:07 PM  <DIR>          java
02/28/2010  01:31 AM                210 regexJava.txt
02/27/2010  10:23 PM                1,567 RegexTestHarness.class
02/27/2010  10:23 PM                1,276 RegexTestHarness.java
04/06/2010  05:36 PM            134,144 sudoku.doc
04/06/2010  05:28 PM            36,842 sudoku1.jpg
04/06/2010  05:28 PM            35,887 sudoku2.jpg
03/19/2010  10:39 AM            32,256 TestLOR.doc
01/19/2010  02:05 PM                2,056 Twitter.class
01/19/2010  02:03 PM                1,618 Twitter.java
               11 File(s)              247,105 bytes
               4 Dir(s)  284,529,577,984 bytes free

C:\TEMP>
```

Figura 19.2 Uso del comando `dir`.

en la Figura 19.2. Los directorios se indican mediante la palabra `DIR` encerrada entre paréntesis angulares. Para restringir la lista de manera que sólo aparezcan directorios, escriba:

```
dir /ad
```

El comando va seguido de un espacio, y pueden utilizarse letras mayúsculas o minúsculas. Este comando es útil para recordarnos los nombres de los directorios a medida que navegamos.

### El comando `cd`

Este comando se utiliza para desplazarse de un directorio a otro. Cabe mencionar que los caracteres `..` se emplean para indicar el directorio inmediatamente superior del actual. He aquí algunos ejemplos.

```
cd midir      cambia al subdirectorio midir
cd ..        se desplaza un nivel hacia arriba
cd ../datos  se desplaza hacia arriba y luego hacia el directorio datos
```

Observe que al cambiar de directorio el indicador de línea de comandos también cambia. Use esto para comprobar que su comando haya funcionado de manera apropiada. Por último, para desplazarse a otra unidad escriba la letra que la identifica, de la siguiente forma:

```
d:\          cambia a la unidad d:
```

#### PRÁCTICA DE AUTOEVALUACIÓN

**19.1** Averigüe cómo ejecutar la ventana del símbolo del sistema en su computadora. Experimente con los comandos `dir` y `cd`. Navegue hasta el directorio de un proyecto de C#, y encuentre el archivo `exe` que está dentro del directorio `Debug`, que a su vez se encuentra en la carpeta `bin`.

### ● Formas de ejecutar programas

Hay varias formas de iniciar la ejecución de programas de C# en el entorno Windows. He aquí algunas de ellas:

- Desde el IDE; éste es el método que se utiliza al crear y depurar programas.
- Haciendo doble clic sobre su nombre en el Explorador de Windows. Por ejemplo, el programa `Hola` que creamos al principio del capítulo se guardó como `Hola.cs`, y también puede ejecutarse si navegamos hasta el directorio `Debug` del proyecto correspondiente y hacemos doble clic en `Hola.exe` (nota: dependiendo de la versión de Windows que utilice y de su configuración, tal vez sólo necesite hacer un clic para ejecutar los programas).
- Escribiendo su nombre en el símbolo del sistema. Por ejemplo, para ejecutar nuestro programa `Hola` abrimos la ventana de símbolo del sistema, navegamos hasta el directorio `Debug` (dentro

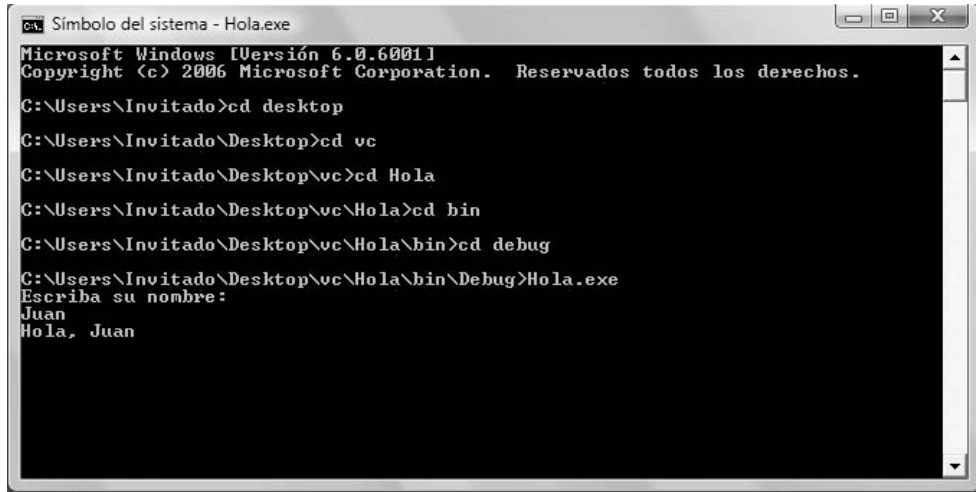


Figura 19.3 Ejecución del programa `hola` desde la ventana del símbolo del sistema.

de la carpeta `bin`) del proyecto mediante el uso del comando `cd`, y después escribimos el nombre del archivo `exe`, como en el siguiente ejemplo:

```
Hola.exe
```

La Figura 19.3 muestra la ventana del símbolo del sistema después de ejecutar el programa. Observe que la ventana no se cierra; está lista para que escribamos otro comando.

Si desea ejecutar el programa estando en otro directorio, puede escribir la ruta completa para el archivo `exe`. Por ejemplo, si el directorio del proyecto se guarda en el nivel superior de la unidad `c:`, puede escribir lo siguiente:

```
c:\Hola\bin\Debug\Hola.exe
```

- También puede ejecutar un programa “por lotes”. He aquí un ejemplo, el cual puede ejecutar siguiendo estos pasos:
  1. Utilice un editor de texto (como el **Bloc de notas**, que encontrará en la carpeta **Accesorios**) para crear un archivo llamado `hola.bat` (si el editor no tiene opción para guardarlo con la extensión `.bat`, puede guardarlo como `.txt` y después renombrarlo a `.bat`). Dentro del archivo agregue estas dos líneas de texto:

```
c:\Hola\bin\Debug\Hola.exe  
c:\Hola\bin\Debug\Hola.exe
```

Guarde el archivo en un directorio de su elección. Puede hacerlo en `c:\temp`.

2. Ejecute el archivo `bat` haciendo doble clic en `hola.bat` en el Explorador de Windows. Observe que el sistema le pedirá su nombre dos veces.

Como acaba de ver, los archivos por lotes contienen instrucciones para ejecutar una serie de programas. En nuestro caso elegimos ejecutar el mismo programa dos veces. Más adelante veremos esta metodología con más detalle.

## ● Uso de clases en aplicaciones de consola

---

En el caso de programas pequeños tal vez esté bien usar unos cuantos métodos estáticos, como en el siguiente ejemplo:

```
static void Main(string[] args)
{
    MétodoA();
}

private static void MétodoA()
{
    // ... código para el método
}
```

Sin embargo, para programas más grandes quizá sea conveniente incorporar clases de biblioteca. Para ello podemos colocar la instrucción `using` apropiada en la parte superior del archivo.

Es probable que también sea necesario crear nuevas clases. Al igual que con los programas de GUI, puede colocar las clases en archivos separados o dentro del mismo archivo.

## ● Argumentos de la línea de comandos

---

Al ejecutar aplicaciones de consola suele ser necesario proveer los detalles iniciales (como los nombres de archivo) para poder utilizar ciertos elementos. Por ejemplo, el programa podría utilizar el método `ReadLine` para recibir como entrada dichos elementos, y ésta es una buena solución en algunos casos. Sin embargo, más adelante veremos programas que operan sin interacción por parte del usuario, pero que de todas formas necesitan cierto tipo de inicialización. Aquí es donde entran en acción los argumentos de la línea de comandos.

Veamos un ejemplo: escribiremos un programa llamado `Elegir`, el cual busca a través de un archivo específico todas las líneas que contengan una cadena de caracteres en particular, y las despliega en pantalla. El código es el siguiente:

```
static void Main(string[] args)
{
    string línea;
    string nombreArchivo = args[0];
    string buscado = args[1];
    StreamReader flujoEntrada = File.OpenText(nombreArchivo);

    línea = flujoEntrada.ReadLine();
    while (línea != null)
    {
        if (línea.IndexOf(buscado) >= 0)
        {
            Console.WriteLine(línea);
        }
        línea = flujoEntrada.ReadLine();
    }
    flujoEntrada.Close();
    String espera = Console.ReadLine();
}
```

El programa lee por separado cada línea del archivo especificado (como vimos en el capítulo 18), y utiliza el método `IndexOf` de la clase `String` para determinar si existe la cadena requerida en cualquier parte de la línea. De ser así, despliega la línea en la pantalla.

Cabe mencionar que el programa debe obtener los valores para las cadenas `nombreArchivo` y `buscado`. En un momento dado las utilizaremos de la manera convencional mediante la ventana del símbolo del sistema, pero para fines de depuración y prueba el IDE nos permite establecer esos valores. He aquí cómo:

1. Estando en el IDE, cree un proyecto Aplicación de consola llamado `Elegir`, y escriba el código de C# anterior. Guarde el código en el archivo `Elegir.cs`.
2. A continuación estableceremos los argumentos de la línea de comandos. Vaya a la ventana **Explorador de soluciones** y haga clic con el botón derecho del ratón en el nombre del proyecto. Seleccione **Propiedades**. A continuación aparecerá la **Página de propiedades**.
3. Seleccione la opción **Depurar**. Escriba lo siguiente en el área **Argumentos de la línea de comandos**:

```
c:\temp\demo.txt "lineas"
```

Haga clic en **Aceptar**. Las cadenas para los argumentos van separadas por uno o varios espacios. Puede usar comillas alrededor de un argumento si lo desea; esto es esencial si el argumento en sí contiene espacios. En este ejemplo específico elegimos `c:\temp\` para evitar problemas, en caso de que trabaje en una red que controle estrictamente los directorios en los que pueda crear archivos. Si éste no es su caso, puede seleccionar cualquier otro directorio.

Si desea omitir la ruta del archivo y especificar sólo `demo.txt`, C# asumirá que el archivo se encuentra en el mismo directorio que `Elegir.exe`; es decir, en `bin\Debug`. Si omite la ruta, recuerde no omitir la palabra “lineas”, que en este caso es la palabra que se está buscando.

4. Cree un archivo llamado `demo.txt` que contenga estas líneas:

```
Estas lineas son
una prueba para el programa Elegir.
El programa encuentra y despliega las lineas
que contengan una subcadena especifica.
```

Guarde este archivo en `c:\temp\` o en `bin\Debug`.

Cuando ejecute el programa, los resultados aparecerán en una ventana negra que se cerrará casi de inmediato. Para ver los resultados coloque la siguiente línea justo antes de la llave de cierre `}` de `Main`, como hicimos en el programa `Hola`:

```
string espera = Console.ReadLine();
```

Ejecute el programa de nuevo y verá dos líneas como las siguientes, cada una de las cuales contiene la cadena `lineas`:

```
Estas lineas son
El programa encuentra y despliega las lineas
```

Oprima la tecla **Enter** para finalizar el programa.

Al ejecutar una aplicación de consola el sistema, en tiempo de ejecución, invoca el método `Main`. Nosotros no lo invocamos. Cualquier argumento de la línea de comandos que proporcionemos se

pasa al método como un arreglo unidimensional de cadenas de caracteres llamada `args`. Antes de colocar los argumentos en este arreglo se lleva a cabo cierto procesamiento de manera automática:

- Los argumentos se dividen y se separan mediante uno o varios espacios.
- Cualquier espacio encerrado entre comillas no se considera separador. Por ejemplo, en nuestro programa podemos tener rutas de archivos que contengan espacios, o tal vez necesitemos buscar una cadena que los incluya. En estos casos debemos establecer los argumentos encerrados entre comillas. Las comillas se eliminan de cada argumento antes de pasarlo al método `Main`.
- El primer argumento se coloca en la posición 0 de nuestro arreglo, no en la posición 1.

Podemos utilizar la propiedad `Length` del arreglo para averiguar cuántos argumentos proporcionó el usuario y rechazar cualquier error.

#### PRÁCTICAS DE AUTOEVALUACIÓN

**19.2** Corrija el programa `Elegir`, de manera que compruebe el número de argumentos proporcionados. Si no hay dos argumentos, el programa deberá mostrar un mensaje de error y terminar.

**19.3** Modifique el programa `Elegir` de manera que encuentre versiones en minúsculas y mayúsculas de la cadena requerida.

## ● Secuencias de comandos y redirección de la salida

En esta sección veremos parte de las herramientas que ofrecen los sistemas Windows para las secuencias de comandos, y utilizaremos algunas de ellas en nuestro programa `Elegir`. El uso de secuencias de comandos cubre diversas técnicas. En nuestro caso consideraremos una secuencia de comandos como un archivo que contiene instrucciones para ejecutar una serie de programas. A estos archivos se les conoce como archivos de procesamiento por lotes, y tienen la extensión `.bat`. Su uso fomenta la metodología del “bloque de construcción”, en donde para crear nuevo software conectamos las aplicaciones de software existentes entre sí, en vez de codificar desde cero.

La redirección de la salida nos permite controlar el destino del texto de salida de un programa. En nuestros programas anteriores utilizamos `Console.WriteLine` para enviar texto a la pantalla. Sin embargo, algunas veces es conveniente colocar los resultados en un archivo. He aquí un ejemplo: necesitamos colocar la salida de `Elegir` en un archivo, para poder utilizar los datos en cualquier otra parte. Para ello podemos abrir una ventana de símbolo del sistema y utilizar el comando `cd` para ir al directorio `Debug` del proyecto `Elegir`. Escriba lo siguiente:

```
Elegir.exe c:\temp\demo.txt "lineas" > c:\temp\resultados.txt
```

El uso de `>` hace que el texto que comúnmente aparece en pantalla se redirija al archivo `resultados.txt`. Emplee un editor de texto para examinar el contenido de `resultados.txt`, ubicado en

el directorio `c:\temp`. Si este archivo ya existiera antes de ejecutar la línea anterior, se sobrescribiría con los nuevos datos. Si no existiera, se crearía automáticamente.

Decidimos colocar espacios alrededor del signo `>` para facilitar la legibilidad del ejemplo, aunque esto no es esencial.

Otra cuestión de preferencia es el uso de mayúsculas. El software de la línea de comandos se basa en MS-DOS, uno de los primeros sistemas operativos de Microsoft. Los nombres de archivo y las rutas se convierten internamente a mayúsculas tan pronto como los escribimos, por lo que se obtiene el mismo efecto con cualquiera de las siguientes líneas:

```
ELEGIR.EXE C:\TEMP\DEMO.TXT "lineas"
Elegir.exe c:\temp\demo.txt "lineas"
elegir.exe c:\temp\demo.txt "lineas"
```

He aquí otro ejemplo: deseamos encontrar todas las líneas que contengan una instrucción `if` en un programa de C#, suponiendo que el programa tenga sangría y que exista un espacio entre la instrucción `if` y el signo `(`. Además deseamos agregar estas líneas al archivo `resultados.txt` que creamos antes. Para ello escribimos la siguiente línea:

```
Elegir.exe c:\temp\UnProg.cs " if (" >> c:\temp\resultados.txt
```

El símbolo `>>` significa “adjuntar” y su acción es muy parecida a la de `>`, sólo que el texto se coloca al final de un archivo existente en vez de reemplazar su contenido.

También es posible redirigir la entrada mediante el signo `<`, pero dicho objetivo requiere una técnica distinta para leer los datos y enviarlos como entrada al programa. En nuestros ejemplos siempre leeremos el contenido de los archivos en vez de redirigir la entrada.

El beneficio de redirigir la salida es, que cambiar el destino de los datos será un proceso simple: no necesitamos volver a escribir el programa ni hacer innumerables clics para navegar por un cuadro de diálogo. El programa se vuelve más flexible.

## PRÁCTICAS DE AUTOEVALUACIÓN

**19.4** Si abrimos una ventana de símbolo del sistema, nos cambiamos al directorio `bin\Debug` del proyecto `Hola` y escribimos:

```
Hola.exe > c:\temp\temp.txt
```

¿Qué esperaríamos ver en pantalla?, ¿cuáles serían los resultados en el archivo `temp.txt`?

**19.5** Éstos son dos casos en los que usamos el programa `Elegir`. En el primero utilizamos `>`:

```
Elegir.exe archivoprueba.txt "C#" > c:\temp\salida.txt
Elegir.exe archivoprueba.txt "C#" > c:\temp\salida.txt
```

En el segundo empleamos `>>`:

```
Elegir.exe archivoprueba.txt "C#" >> c:\temp\salida.txt
Elegir.exe archivoprueba.txt "C#" >> c:\temp\salida.txt
```

¿De qué manera afecta el uso de `>` o `>>` al contenido del archivo `salida.txt`?

## ● Secuencias de comandos y archivos de procesamiento por lotes

Antes de que existieran las computadoras personales se utilizaban tarjetas perforadas: el desarrollador tenía que preparar un “lote” de tarjetas que contenían el programa, además de las instrucciones sobre cómo ejecutarlo. Este concepto de secuencia de instrucciones tiene su equivalente moderno en el archivo `bat`, mismo que nos permite preparar un archivo que contiene instrucciones para ejecutar varios programas sin interacción del usuario.

Veamos un ejemplo: como parte de la documentación de un programa de gran tamaño construido en C# (al que llamaremos `Gran.cs`), deseamos crear un archivo que contenga todas las declaraciones `public` y `private`. Para ello necesitamos utilizar el programa `Elegir` dos veces. Podemos hacerlo creando primero un archivo llamado (por ejemplo) `decs.bat`, ubicado en un directorio adecuado, digamos `c:\temp\`. El archivo contiene lo siguiente:

```
c:\proyectos\Elegir\bin\Debug\Elegir.exe Gran.cs "private" > decs.txt
c:\proyectos\Elegir\bin\Debug\Elegir.exe Gran.cs "public" > decs.txt
```

Después ejecutamos el archivo. Podemos hacerlo de varias formas, como vimos antes: haciendo doble clic sobre su nombre en el Explorador de Windows, o escribiendo su nombre en el símbolo del sistema. En cualquier caso se inicia la ejecución de las instrucciones que están dentro del archivo. El primer uso de `Elegir` coloca todas las líneas que contengan `private` dentro del archivo `decs.txt`, y el segundo adjunta todas las líneas que contengan la cadena `public` al final del mismo archivo. Las instrucciones para ejecutar `Elegir` dentro del archivo de procesamiento por lotes se obedecen en orden, de arriba hacia abajo.

El ejemplo anterior utiliza los directorios de manera muy distinta a como lo habíamos venido haciendo. Suponga que el archivo `exe` (en nuestra computadora) está almacenado en:

```
c:\proyectos\Elegir\bin\Debug\
```

Como el archivo `bat` está en otra parte (por ejemplo, en `c:\temp\`), debemos proveer a `Elegir.exe` la ruta completa dentro del archivo `bat`.

Se asume que los archivos `decs.txt` y `Gran.cs` se encuentran en el directorio desde el que se ejecuta `Elegir`. Éste es `c:\temp\`, no `c:\proyectos\Elegir\bin\Debug\`. Si tenemos estos archivos en otra parte, podemos indicar las rutas completas en el archivo `bat`.

### PRÁCTICA DE AUTOEVALUACIÓN

**19.6** Suponga que recibimos un gran programa creado en C# y deseamos encontrar todas las líneas que contengan una instrucción `if` que haga referencia a una variable llamada `suma`. Por ejemplo, queremos líneas tales como:

```
if (suma == 0)
if (a == suma)
```

Prepare un archivo de procesamiento por lotes que utilice el programa `Elegir` para realizar la tarea. Sugerencia: use un archivo temporal llamado (por ejemplo) `temp.txt`.



## Fundamentos de programación

- No todos los programas justifican una GUI.
- La redirección de entrada/salida puede hacer que los programas sean más flexibles.
- Una solución con un archivo de procesamiento por lotes puede evitar la necesidad de escribir nuevo software.

## Errores comunes de programación

- Olvidar poner una invocación a `ReadLine` cuando deseamos observar la salida en pantalla.
- Acceder a los argumentos de la línea de comandos desde la posición 1 en el arreglo de cadenas. De hecho, los elementos empiezan en la posición 0.

## Secretos de codificación

En este capítulo no se presentó nueva gramática de C#.

## Nuevos elementos del lenguaje

- La clase `Console` y los métodos `ReadLine`, `WriteLine` y `Write`.

## Nuevas características del IDE

- Existe una opción para crear aplicaciones de consola.
- Se pueden establecer los argumentos de la línea de comandos dentro del IDE, para fines de prueba y depuración.

## Resumen

Las aplicaciones de consola pueden:

- mostrar líneas de texto en pantalla, y leer las líneas de texto escritas por el usuario.
- acceder a los argumentos de la línea de comandos.
- ejecutarse de varias formas.
- redirigir su salida a un archivo mediante los signos `>` y `>>`.
- Los archivos de procesamiento por lotes pueden contener instrucciones para ejecutar programas.

## EJERCICIOS

- 19.1** Escriba una aplicación de consola que pida al usuario introducir un entero. La aplicación deberá calcular el doble del valor y desplegar el resultado en pantalla (suponga que el usuario no cometerá errores al introducir el número).
- 19.2** Escriba una aplicación de consola que reciba como entrada del usuario una serie de calificaciones de exámenes (una a la vez). El programa deberá mostrar la suma de todas las calificaciones. Éstas pueden ser iguales o mayores a cero; se utilizará un número negativo para terminar la secuencia de calificaciones (emplee un ciclo `while`).
- 19.3** Escriba una aplicación de consola que muestre la primera línea de un archivo. Debe especificar el nombre del archivo como argumento de la línea de comandos. Compruebe que el resultado correcto aparezca en pantalla, y después ejecútelo utilizando redirección de salida para colocar los resultados en un archivo.
- 19.4** Escriba una aplicación de consola que cuente el número de líneas de que consta un archivo, y despliegue el resultado.
- 19.5** Hay quienes sugieren que podemos medir la complejidad de un programa si contamos el número de decisiones y ciclos de que está compuesto. El ideal es crear programas que no sean demasiado complejos. Con la ayuda de nuestro programa `Elegir` y de la solución que provea para el ejercicio 19.4, cree un archivo de procesamiento por lotes que sume el número de instrucciones `if`, `while`, `for`, `do` y `switch` en un archivo de C# y muestre el resultado (no escriba código nuevo de C#).
- 19.6** Escriba una aplicación de consola que lea un archivo línea por línea y lo despliegue en pantalla. Sin embargo, las líneas que contengan una subcadena específica no deberán mostrarse. El programa debe tener dos argumentos de línea de comandos: el nombre de archivo y la subcadena a localizar. Este último argumento puede encerrarse entre comillas por si contiene espacios. Use el programa para proporcionar una versión “colapsada” de un programa de C#, omitiendo todas las líneas que contengan ocho espacios (esto hará que se ignore el código anidado interno, y que se muestren las líneas externas como `using`, `class` y `private`).
- 19.7** Escriba una aplicación de consola que reemplace una cadena por otra, a lo largo de todo un archivo. El programa debe tener tres argumentos: el nombre del archivo, la cadena a reemplazar y la cadena con la que se va a realizar el reemplazo. Lea todo el archivo en una sola operación y colóquelo en una cadena de texto; utilice el método `Reemplazar` que creamos en el capítulo 16. Redirija la salida a un nuevo archivo.
- 19.8** Escriba un programa llamado `Multiremp` que realice varios reemplazos de cadenas de caracteres en un archivo. Los argumentos para el programa deben ser: el nombre del archivo de entrada y el nombre de un archivo que contenga pares de líneas que especifiquen cada una de las líneas que se van a reemplazar, y cada una de las líneas que servirán como reemplazo. He aquí un ejemplo. Queremos reemplazar todas las ocurrencias de `sr` por `sra`, y todas las ocurrencias de `él` por `ella` en el archivo `datos.txt`. Para ejecutar el programa escribimos lo siguiente:

```
Multiremp datos.txt cambios.txt > salida.txt
```

El archivo `cambios.txt` contiene:

```
Sr
Sra
él
ella
```

Si conoce otro lenguaje de programación, investigue el uso de `Multitemp` como ayuda para convertir un programa de C# a ese lenguaje.

## SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

**19.1** Se necesita un poco de práctica. Recuerde ver el indicador de línea de comandos para saber cuál es el directorio actual, y escriba `dir` para comprobar la escritura exacta de los nombres de los directorios.

**19.2** Agregamos una instrucción `if` al código, y también nos aseguramos de no acceder a los argumentos antes de comprobar que existan.

```
string línea;
string nombreArchivo;
string buscado;
if (args.Length == 2 )
{
    nombreArchivo = args[0];
    buscado = args[1];
    StreamReader flujoEntrada = File.OpenText(nombreArchivo);
    línea = flujoEntrada.ReadLine();
    while (línea != null)
    {
        if (línea.IndexOf(buscado) >= 0)
        {
            Console.WriteLine(línea);
        }
        línea = flujoEntrada.ReadLine();
    }
    flujoEntrada.Close();
}
else
{
    Console.WriteLine("Error: se necesitan dos argumentos para Elegir");
}
```

**19.3** Convertimos ambas cadenas de texto a mayúsculas antes de compararlas, como en el siguiente ejemplo:

```
if (línea.ToUpper().IndexOf(buscado.ToUpper()) >= 0)
```

- 19.4** El texto de `Write` y `WriteLine` no aparece en pantalla, por lo que tenemos que suponer que debemos escribir nuestro nombre. En el archivo vemos lo siguiente:

```
Escriba su nombre: Hola Juan
```

- 19.5** En el caso de `>`, el archivo sólo contendrá las líneas que tengan la cadena requerida. El segundo uso de `>` sobrescribirá el contenido anterior.

En el caso de `>>`, el archivo contiene las mismas líneas duplicadas. Además, si el archivo de salida no está vacío, se conservará su contenido original.

- 19.6** La primera ejecución de `Elegir` deberá seleccionar todas las líneas con `if`. Estas líneas se usarán entonces como entrada para otra ejecución de `Elegir`, en donde se escogerán las líneas que contengan `suma`. He aquí el código, que podemos colocar en un archivo llamado `ifsuma.bat`:

```
c:\proyectos\Elegir\bin\Debug\Elegir.exe Gran.cs " if (" > temp.txt  
c:\proyectos\Elegir\bin\Debug\Elegir.exe temp.txt "suma" > salida.txt
```

Las líneas seleccionadas se redirigieron a un archivo llamado `salida.txt`.

# El diseño orientado a objetos

**En este capítulo conoceremos cómo:**

- identificar las clases que se necesitan en un programa;
- diferenciar entre la composición y la herencia;
- utilizar algunos lineamientos para el diseño de clases.

## ● Introducción

---

Para empezar a diseñar un puente es probable que no se necesite pensar en el tamaño de los remaches. Primero habría que tomar decisiones importantes, como si el puente debe ser colgante o suspendido. De manera similar, no empezaría el diseño de un edificio pensando en el color de las alfombras; primero tomaría determinaciones fundamentales, como cuántos pisos debe tener y en dónde deben estar los elevadores. Partiendo de esta analogía, a lo largo del capítulo le explicaremos cómo utilizar un método establecido para diseñar programas orientados a objetos.

Los programas pequeños en realidad no requieren diseño; podemos simplemente crear la interfaz de usuario y proceder de inmediato a escribir las instrucciones de C#. Pero es bien sabido que, en el caso de programas grandes, el desarrollador debe empezar por tomar las decisiones importantes en vez de ocuparse de los detalles; es decir, debe iniciar su labor realizando un buen diseño, y posponer cuestiones como el formato exacto de un número o la posición de un botón. Desde luego, todas las etapas de la programación son cruciales, pero algunas lo son más que otras.

Cuando empezamos a escribir programas, por lo general invertimos mucho tiempo en el procedimiento de prueba y error; esto suele ser muy divertido y creativo. A veces también pasamos cierto tiempo luchando con el lenguaje de programación, pues se requiere tiempo para aprender las buenas prácticas y reconocer las malas. Se necesita aún más tiempo para adoptar una metodología de diseño efectiva; una vez logrado este objetivo, la diversión y la creatividad permanecen, pero las partes molestas de la programación se reducen.

El proceso de diseño recibe como entrada la especificación de lo que el programa debe hacer. El producto final del proceso de diseño es una descripción de las clases, propiedades y métodos que empleará el programa.

Utilizaremos el ejemplo sencillo del programa del globo para ilustrar la fase del diseño. También introduciremos un ejemplo de diseño más complejo.

## ● **El problema del diseño**

---

Hemos comentado con anterioridad que los programas orientados a objetos consisten en una colección de objetos. Al empezar a desarrollar un nuevo programa el problema estriba en identificar cuáles son objetos apropiados. Una vez que hayamos identificado los objetos, cosecharemos todos los beneficios de la programación orientada a objetos (POO). Ahora bien, la dificultad fundamental de la POO radica en cómo identificar los objetos. Esto es lo que ofrece un método de diseño: la metodología o serie de pasos para lograr dicho objetivo. Es como cualquier otro tipo de diseño: requiere un método. No basta conocer los fundamentos de la programación orientada a objetos. Como analogía, conocer las leyes de la física no significa que podamos diseñar una nave espacial; también hay que llevar a cabo cierto proceso de diseño.

Uno de los principios utilizados en el diseño de los programas orientados a objetos es simular las situaciones del mundo real como objetos. Construimos un modelo de software de las cosas que existen en el mundo real. He aquí algunos ejemplos:

- Si vamos a desarrollar un sistema de automatización de oficinas, debemos simular los usuarios, el correo, los documentos compartidos y los archivos.
- En un sistema de automatización industrial tendríamos que simular las distintas máquinas, las líneas de producción, los pedidos y las entregas.

Así, la metodología es identificar los objetos que participan en el problema que buscamos resolver, y modelarlos como objetos en el programa.

La abstracción juega un papel en este proceso. Sólo necesitamos modelar las partes relevantes de los problemas a resolver, por lo cual podemos ignorar cualquier detalle irrelevante. Si modelamos un globo necesitamos representar su posición, su tamaño y su color, pero no es necesario modelar el material de que está hecho. Si vamos a crear un sistema de registros de personal, probablemente tengamos que modelar los nombres, las direcciones y las descripciones de trabajo, pero no los pasatiempos ni los estilos de música preferidos por los empleados.

## ● **Identificación de los objetos, métodos y propiedades**

---

Una manera efectiva de llevar a cabo el diseño orientado a objetos consiste en examinar la especificación de software para extraer información sobre los objetos, propiedades y métodos. La metodología para identificar los objetos y métodos es:

1. Buscar sustantivos (nombres de cosas) en la especificación; éstos son los objetos.
2. Buscar verbos (palabras que indiquen acción) en la especificación; éstos son los métodos.

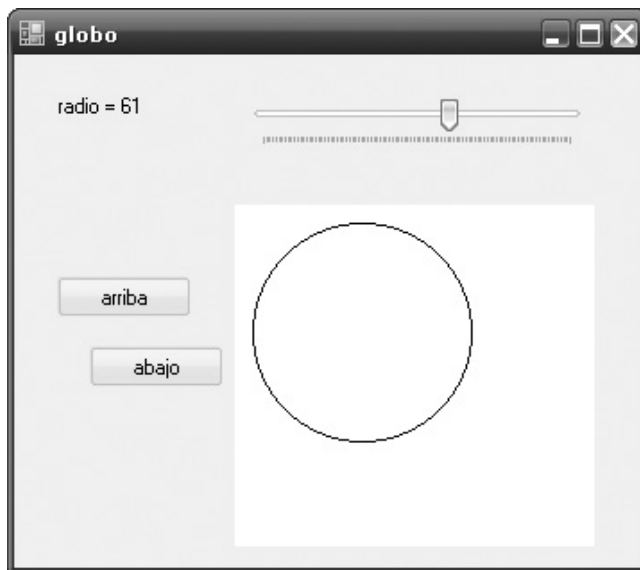


Figura 20.1 El programa del globo.

Por ejemplo, a continuación le mostramos la especificación para el programa simple del globo:

Escriba un programa para representar un globo y manipularlo mediante una GUI. El globo debe aparecer como un círculo dentro de un cuadro de imagen. Utilice botones para cambiar la posición del globo desplazándolo una distancia fija hacia arriba o hacia abajo. Use una barra de seguimiento para modificar el radio del globo, el cual debe desplegarse en una etiqueta.

El formulario correspondiente se muestra en la Figura 20.1.

De acuerdo con lo que se señaló antes, debemos buscar verbos y sustantivos en la especificación. En este caso podemos identificar los siguientes sustantivos:

GUI, cuadro de imagen, botón, barra de seguimiento, etiqueta, globo, posición, distancia, radio

La GUI provee la interfaz de usuario para el programa. Consiste en botones, una barra de seguimiento, una etiqueta y un cuadro de imagen. La GUI se representa mediante un objeto que es una instancia de la clase `Form1`. Los objetos botón, barra de seguimiento, etiqueta y cuadro de imagen están disponibles como clases en la biblioteca de C#.

El objeto GUI:

- crea en pantalla los botones, la barra de seguimiento, la etiqueta y el cuadro de imagen;
- maneja los eventos de los clics de ratón en los botones y la barra de seguimiento;
- crea cualquier otro objeto que se necesite, como el objeto globo;
- invoca los métodos y utiliza las propiedades del objeto globo.

El siguiente objeto importante es el globo, que utiliza cierta información para representar su posición (coordenadas  $x$  y  $y$ ), la distancia que se desplaza y su radio. Una opción sería crear varios objetos completos para representar estos elementos, pero es más simple representarlos como variables `int`.

Con esto completamos la identificación de los objetos dentro del programa. Nuestro siguiente trabajo es generalizar los objetos y diseñar las clases que corresponden a cada uno de ellos. Por lo tanto necesitamos las clases `GUI`, `Label`, `Globo`, etc.

Ahora extraeremos los verbos de la especificación:

`CambiarRadio`, `MoverArriba`, `MoverAbajo`, `MostrarGlobo`, `MostrarRadio`

Debemos crear los métodos correspondientes dentro del programa que estamos diseñando; y necesitamos decidir a cuál objeto pertenecen: al objeto `GUI` o al objeto globo. Parece razonable que los verbos `MoverArriba`, `MoverAbajo` y `MostrarGlobo` son métodos asociados con el objeto globo.

Es momento de concentrar nuestra atención en los verbos `CambiarRadio` y `MostrarRadio`. Ya hemos decidido que el valor del radio se implementará como una variable `int` dentro del objeto `Globo`. Sin embargo, el objeto `GUI` necesita tener acceso a ese valor para desplegarlo en la etiqueta. También es necesario que cambie el valor en respuesta a las modificaciones del valor de la barra de seguimiento. Por lo tanto, la clase `Globo` necesita proveer acceso al valor del radio, lo cual podemos lograr utilizando ya sea:

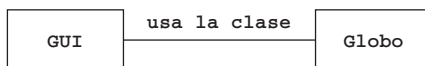
- métodos (llamados `GetRadio` y `SetRadio`), o
- una propiedad (llamada `Radio`).

En este caso elegimos la segunda opción.

Para resumir, nuestro diseño para este programa consiste en dos clases que no son de biblioteca: `GUI` y `Globo`, las cuales se muestran en el diagrama de clases UML (Figura 20.2). Este diagrama ilustra las principales clases que intervienen en el programa, y sus interrelaciones. La clase `GUI` utiliza la clase `Globo` mediante invocaciones a sus métodos, y además emplea su propiedad.

Podemos documentar cada clase mediante diagramas de clases más detallados. En estos diagramas cada cuadro describe una sola clase y consta de cuatro secciones que proporcionan información sobre:

1. El nombre de la clase.
2. Una lista de variables de instancia.
3. Una lista de métodos.
4. Una lista de propiedades.



**Figura 20.2** Diagrama de clases que muestra las dos clases principales que conforman el programa Globo.



Antes que nada, analicemos la descripción de la clase GUI:

Clase GUI
<b>Variables de instancia</b>  botónArriba botónAbajo trackBar1 label1 pictureBox1
<b>Métodos</b>  botónArriba_Click botónAbajo_Click trackBar1_Scroll

Ahora veamos el diagrama de clase de la clase `Globo`:

Clase Globo
<b>Variables de instancia</b>  coordenadaX coordenadaY radio distancia
<b>Métodos</b>  MoverArriba MoverAbajo Mostrar
<b>Propiedades</b>  Radio

Ahora el diseño de este programa está completo. El diseño termina en el momento en que se especifican todas las clases, objetos, propiedades y métodos. El diseño no tiene injerencia en la escritura (codificación) de las instrucciones de C# que forman estas clases, propiedades y métodos. Sin embargo, es natural que el lector sienta curiosidad sobre el código, por lo cual ahora le mostraremos el correspondiente a la clase `GUI`:

En la parte superior de la clase están las variables de instancia:

```
private Globo globo;
private Graphics áreaDibujo;
```

Dentro del método constructor está la creación del objeto `Globo`, la inicialización del cuadro de imagen y de la etiqueta:

```
globo = new Globo();
áreaDibujo = pictureBox1.CreateGraphics();
radioLabel.Text = "radio = ";
```

Después tenemos los manejadores de eventos:

```
private void botónAbajo_Click(object sender, EventArgs e)
{
    globo.MoverAbajo();
    Dibujar();
}

private void botónArriba_Click(object sender, EventArgs e)
{
    globo.MoverArriba();
    Dibujar();
}

private void trackBar1_Scroll(object sender, EventArgs e)
{
    globo.Radio = trackBar1.Value;
    Dibujar();
}
```

Y un método útil:

```
private void Dibujar()
{
    radioLabel.Text = "radio = " + Convert.ToString(globo.Radio);
    globo.Mostrar(áreaDibujo);
}
```

A continuación le mostramos el código de la clase `Globo`:

```
public class Globo
{
    private int x = 50;
    private int y = 50;
    private int radio = 20;
    private int yIntervalo = 20;
    Pen lápiz = new Pen(Color.Black);

    public void MoverArriba()
    {
        y = y - yIntervalo;
    }
}
```

```
public void MoverAbajo()
{
    y = y + yIntervalo;
}

public void Mostrar(Graphics áreaDibujo)
{
    áreaDibujo.Clear(Color.White);
    áreaDibujo.DrawEllipse(lápiz, x, y, radio * 2, radio * 2);
}

public int Radio
{
    get
    {
        return radio;
    }
    set
    {
        radio = value;
    }
}
}
```

Éste es un programa simple, con sólo dos objetos que no son de biblioteca. Sin embargo, ilustra cómo extraer objetos, métodos y propiedades de una especificación. Más adelante analizaremos un ejemplo más complejo.

Para sintetizar, el método de diseño para identificar métodos y objetos consiste en:

1. Buscar sustantivos en la especificación; éstos son los objetos (o algunas veces simples variables).
2. Buscar verbos en la especificación; éstos son los métodos.

Una vez que identificamos los objetos, es muy sencillo generalizarlos y convertirlos en clases.

Cabe mencionar que aunque este programa está diseñado con dos clases, podríamos haberlo diseñado con una sola. No obstante, el diseño que mostramos antes hace un uso mucho más explícito de los objetos presentes en la especificación del programa. El diseño también separa la parte del programa correspondiente a la GUI de la clase globo. Ésta es una estructura de programa ampliamente recomendada, en donde se separan el código de presentación y el modelo (al que algunas veces se le denomina lógica de dominio). A esta estructura suele llamársele (por razones históricas) arquitectura modelo-vista-controlador. Esto nos permite modificar un programa con más facilidad, ya que la GUI puede modificarse de manera independiente a la lógica interna. Por ejemplo, suponga que deseamos agregar otra barra de seguimiento a la GUI para controlar la posición del globo. Sin duda necesitamos realizar una pequeña modificación a la clase `GUI`, pero para ello no es necesario modificar la clase `Globo`.



Figura 20.3 El programa Invasor del ciberespacio.

### ● Ejemplo práctico de diseño

---

La siguiente es la especificación para crear un programa mucho más grande:

#### *Invasor del ciberespacio*

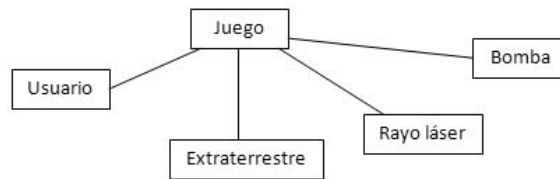
El cuadro de imagen (Figura 20.3) muestra la representación del usuario y un extraterrestre. El extraterrestre se desplaza de un lado a otro. Cuando choca con una pared, invierte su dirección. El extraterrestre lanza una bomba al azar, que se desplaza verticalmente hacia abajo, pero sólo hay una bomba en cualquier momento dado. Si la bomba le pega al usuario, éste pierde. El usuario se desplaza hacia la izquierda o hacia la derecha, de acuerdo con el movimiento que realice con el ratón. Al hacer clic el usuario lanza un rayo láser que se desplaza hacia arriba, pero sólo hay un rayo láser en cualquier momento dado. Si el láser le pega al extraterrestre, el usuario gana.

Recuerde que los principales pasos del diseño son:

1. Identificar los objetos buscando sustantivos en la especificación.
2. Identificar los métodos buscando verbos en la especificación.

Después de analizar la especificación encontramos los siguientes sustantivos. Obviamente, algunos de ellos se mencionan más de una vez.

cuadro de imagen, usuario, extraterrestre, pared, bomba, ratón, rayo láser



**Figura 20.4** Las clases que no son de biblioteca involucradas en el programa del juego.

Estos sustantivos corresponden a objetos potenciales, y por ende pueden ser clases dentro del programa. Por lo tanto, traducimos estos sustantivos y los convertimos en los nombres de las clases que conformarán el modelo. El sustantivo **cuadro de imagen** se traduce en la clase **PictureBox**, disponible en la biblioteca. Los sustantivos **usuario** y **extraterrestre** se traducen en las clases **Usuario** y **Extraterrestre**, respectivamente. El sustantivo **pared** no necesita implementarse como clase, ya que puede ajustarse como un detalle dentro de la clase **Extraterrestre**. El sustantivo **bomba** se traduce en la clase **Bomba**. El sustantivo **ratón** no necesita ser una clase, ya que los eventos de clic del ratón pueden manejarse mediante la clase **Form**. Por último, necesitamos una clase **RayoLáser**. En consecuencia, tenemos la siguiente lista de clases que no son de biblioteca:

**Juego, Usuario, Extraterrestre, RayoLáser, Bomba**

Estas clases se muestran en el diagrama de clases de la Figura 20.4. En él se indica que la clase **Juego** utiliza las clases **Usuario**, **Extraterrestre**, **RayoLáser** y **Bomba**.

Aún no hemos completado nuestra búsqueda de los objetos que participarán en el programa. Para poder detectar las colisiones, es preciso que los objetos sepan en dónde se encuentran los demás y qué tan grandes son. Por lo tanto, en la especificación están implícitas las ideas de la posición y el tamaño de cada objeto. Éstas son las coordenadas *x* y *y*, la altura y el ancho de cada objeto. Aunque éstos son objetos potenciales, pueden implementarse simplemente como variables **int** dentro de las clases **Usuario**, **Extraterrestre**, **RayoLáser** y **Bomba**. Podríamos acceder a estos objetos mediante métodos o propiedades, pero en este caso decidimos usar las propiedades llamadas **x**, **y**, **Altura** y **Ancho**.

Un objeto que hemos ignorado hasta ahora en el diseño es el temporizador de la biblioteca de C#, que está configurado para pulsar a pequeños intervalos de tiempo regulares para poder implementar la animación. Cada vez que el temporizador pulsa, los objetos se desplazan, se borra el contenido del cuadro de imagen y se despliegan todos los objetos. Otro de los objetos que aún no hemos considerado es el generador de números aleatorios, el cual se crea a partir de la clase de biblioteca **Random** para controlar cuándo lanza el extraterrestre las bombas.

Hemos terminado de identificar las clases que conforman el programa del juego.

Podemos explorar de nuevo la especificación, esta vez en busca de verbos que podamos adjuntar a la lista de objetos. En esta ocasión podemos ver:

**mostrar, mover, pegar, lanzar, hacer clic, ganar, perder**

De nuevo, algunas de estas palabras se mencionan más de una vez. Por ejemplo, tanto el extraterrestre como el usuario pueden moverse. Además, todos los objetos del juego deben desplegar su imagen en pantalla.

Es momento de asignar métodos a las clases, para lo cual podemos basarnos en la especificación.

Para documentar las clases empleamos un diagrama de clases UML que muestre las variables de instancia, los métodos y las propiedades de cada una de ellas. Empecemos con la clase **Juego**:

clase Juego
Variables de instancia  pictureBox1 animaciónTimer bombaTimer
Métodos  pictureBox1_MouseMove pictureBox1_Click animaciónTimer_Tick bombaTimer_Tick

He aquí el código para esta clase, el cual empieza declarando las variables de instancia:

```
private Graphics papel;  
private Usuario usuario;  
private Extraterrestre extraterrestre;  
private RayoLáser rayoláser;  
private Bomba bomba;
```

Dentro del método constructor de la clase creamos un nuevo objeto cuadro de imagen e invocamos al método `NuevoJuego` para crear nuevos objetos:

```
papel = pictureBox1.CreateGraphics();  
NuevoJuego();
```

Después creamos todos los métodos, incluidos los manejadores de eventos:

```
private void bombaTimer_Tick(object sender, EventArgs e)  
{  
    if (bomba == null)  
    {  
        bomba = new Bomba(extraterrestre.X, extraterrestre.Y);  
    }  
}  
  
private void pictureBox1_Click(object sender, EventArgs e)  
{  
    int xInicial = usuario.X + usuario.Ancho / 2;  
    int yInicial = usuario.Y - usuario.Altura;  
    if (rayoláser == null)  
    {  
        rayoláser = new RayoLáser(xInicial, yInicial);  
    }  
}
```

```
private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
    usuario.Mover(e.X);
    DibujarTodo();
}

private void animaciónTimer_Tick(object sender, EventArgs e)
{
    MoverTodo();
    DibujarTodo();
    ComprobarChoques();
}

public void MoverTodo()
{
    extraterrestre.Mover();
    if (bomba != null)
    {
        bomba.Mover();
    }
    if (rayoláser != null)
    {
        rayoláser.Mover();
    }
}

private void ComprobarChoques()
{
    if (Colisiona(rayoláser, extraterrestre))
    {
        FinJuego("el usuario");
    }
    else
    {
        if (Colisiona(bomba, usuario))
        {
            FinJuego("el extraterrestre");
        }
    }
    if (bomba != null)
    {
        if (bomba.Y > pictureBox1.Height)
        {
            bomba = null;
        }
    }
}
```

```
        if (rayoláser != null)
        {
            if (rayoláser.Y < 0)
            {
                rayoláser = null;
            }
        }
    }

    public bool Colisiona(Sprite uno, Sprite dos)
    {
        if (uno == null || dos == null)
        {
            return false;
        }
        if ( uno.X > dos.X
            && uno.Y < (dos.Y + dos.Altura)
            && (uno.X + uno.Ancho) < (dos.X + dos.Ancho)
            && (uno.Y + uno.Ancho) > (dos.Y))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    private void FinJuego(string ganador)
    {
        rayoláser = null;
        bomba = null;
        animaciónTimer.Enabled = false;
        bombaTimer.Enabled = false;
        MessageBox.Show("Fin del juego: " + ganador + " gana");
        NuevoJuego();
    }

    public void NuevoJuego()
    {
        animaciónTimer.Enabled = true;
        bombaTimer.Enabled = true;
        usuario = new Usuario(imagenUsuario);
        extraterrestre = new Extraterrestre(imagenExtraterrestre);
    }
```



```
private void DibujarTodo()
{
    papel.Clear(Color.White);
    usuario.Dibujar(papel);
    extraterrestre.Dibujar(papel);
    if (rayoláser != null)
    {
        rayoláser.Dibujar(papel);
    }
    if (bomba != null)
    {
        bomba.Dibujar(papel);
    }
}
```

En este diseño la clase `Juego` lleva a cabo una gran parte del trabajo del programa. Por ejemplo, *lanza* un rayo láser y una bomba. También comprueba si el extraterrestre o el usuario han *chocado* con algo. Éstos son verbos que identificamos en el análisis de la especificación anterior.

Consideremos ahora el objeto usuario. Tiene una posición dentro del cuadro de imagen, y un tamaño. Se mueve en respuesta a un movimiento del ratón. Puede mostrarse en pantalla. Por lo tanto, su clase tiene la siguiente especificación:

<b>clase Usuario</b>
<b>Variables de instancia</b>  <b>coordenadaX</b> <b>coordenadaY</b> <b>altura</b> <b>ancho</b>
<b>Métodos</b>  <b>Mover</b> <b>Dibujar</b>
<b>Propiedades</b>  <b>X</b> <b>Y</b> <b>Altura</b> <b>Ancho</b>

A continuación diseñamos la clase `Extraterrestre`. El extraterrestre tiene una posición y un tamaño. Cada vez que el temporizador pulsa, se mueve. Su dirección y velocidad se controlan mediante el tamaño del intervalo que se utiliza al moverlo. Además, puede crearse y mostrarse en pantalla.

<code>clase Extraterrestre</code>
<b>Variables de instancia</b> <code>coordenadaX</code> <code>coordenadaY</code> <code>altura</code> <code>ancho</code> <code>tamIntervalo</code>
<b>Métodos</b> <code>Extraterrestre</code> <code>Mover</code> <code>Dibujar</code>
<b>Propiedades</b> <code>X</code> <code>Y</code> <code>Altura</code> <code>Ancho</code>

Por lo que respecta al objeto rayo láser, tiene una posición y un tamaño; se crea, se mueve y se muestra en pantalla. También debemos comprobar si choca con un extraterrestre.

<code>clase RayoLáser</code>
<b>Variables de instancia</b> <code>coordenadaX</code> <code>coordenadaY</code> <code>altura</code> <code>ancho</code> <code>tamIntervalo</code>
<b>Métodos</b> <code>RayoLáser</code> <code>Mover</code> <code>Dibujar</code>
<b>Propiedades</b> <code>X</code> <code>Y</code> <code>Altura</code> <code>Ancho</code>

Por último, el objeto bomba es muy similar al rayo láser, aunque difieren en que la primera se mueve hacia abajo, mientras que el rayo láser lo hace hacia arriba.

### PRÁCTICA DE AUTOEVALUACIÓN

**20.1** Escriba el diagrama de clase para la clase `Bomba`.

Ahora tenemos la lista completa de clases, junto con los métodos, variables de instancia y propiedades asociadas con cada una de ellas. Además, hemos modelado el juego y diseñamos una estructura para el programa.

## ● En búsqueda de la reutilización

La siguiente tarea de diseño consiste en asegurarnos de no estar reinventando la rueda. Uno de los principales objetivos de la programación orientada a objetos es promover la reutilización de los componentes de software. En esta etapa debemos comprobar si:

- lo que necesitamos está en alguna biblioteca;
- la clase que escribimos el mes pasado pudiera ser útil para lo que necesitamos hoy;
- es posible generalizar algunas de las clases que hemos diseñado para nuestro programa, y obtener una clase más general de la que podamos heredar.

En el programa *Invasor* del ciberespacio podemos hacer buen uso de componentes de GUI como el cuadro de imagen, disponible en la biblioteca de C#. Otros componentes útiles de la biblioteca son el temporizador y el generador de números aleatorios.

Si encuentra una clase existente que sea similar a lo que necesita, tenga en cuenta que puede usar la herencia para personalizarla y lograr que haga lo que usted quiere. En el capítulo 11 vimos cómo escribir el código para obtener la herencia. A continuación analizaremos una metodología para explorar las relaciones entre las clases mediante el uso de las pruebas “es un” y “tiene un”.

## ● ¿Composición o herencia?

Una vez que identificamos las clases que intervendrán en un programa, el siguiente paso es revisar las relaciones entre ellas. Las clases que conforman un programa colaboran entre sí para obtener el comportamiento requerido, pero se utilizan unas a otras de distintas formas. Las clases se relacionan entre sí de dos maneras:

1. Mediante la *composición*: un objeto crea otro a partir de una clase utilizando la palabra clave `new`. Ejemplos de ello son los formularios que crean un botón, o las relaciones que se establecen entre las clases del programa del juego, ilustradas en la Figura 20.4.
2. Mediante la *herencia*: una clase hereda de otra. Un ejemplo es una clase que extiende la clase de biblioteca `Form` (todos los programas que se analizan en este libro lo hacen).

La tarea de diseño más importante estriba en saber diferenciar entre estas dos posibilidades, de manera que podamos aplicar o evitar la herencia. Una forma de comprobar que hemos identificado correctamente las relaciones apropiadas entre las clases, es utilizar la prueba “es un” o “tiene un”:

- El uso de la frase “es un” en la descripción de un objeto (o clase) significa que probablemente éste tiene una relación de herencia (una frase alternativa que tiene el mismo significado es “consiste de”).
- El uso de la frase “tiene un” indica que no hay relación de herencia, sino una relación de composición.

Ahora veamos un ejemplo sobre cómo identificar la herencia. En la especificación de un programa para dar soporte a las transacciones que se realizan en un banco, descubrimos la siguiente información:

Una cuenta bancaria consiste en el nombre de una persona, su dirección, número de cuenta y saldo actual. Hay dos tipos de cuenta: de ahorro y de inversión. Los cuentahabientes tienen que avisar con una semana de anticipación si desean retirar de una cuenta de inversión, pero ésta genera intereses.

Si parafraseamos esta especificación, podríamos decir que “una cuenta de ahorro es una cuenta bancaria” y “una cuenta de inversión es una cuenta bancaria”. Como puede verse, las palabras cruciales son “es una”, y por lo tanto reconocemos que la clase cuenta bancaria es la superclase de las clases cuenta de ahorro y cuenta de inversión. Es decir, las clases cuenta de ahorro y cuenta de inversión, son subclases de la clase cuenta bancaria, por lo que heredan las propiedades y métodos de la superclase; por ejemplo, una propiedad para acceder a la dirección del cuentahabiente, y un método para actualizar su saldo.

En la Figura 20.5 se muestra un diagrama de clases útil para describir las relaciones de herencia.

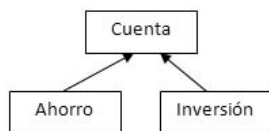


Figura 20.5 Diagrama de clases para las cuentas bancarias.

Considere también el ejemplo de un formulario: “el formulario tiene un botón y un cuadro de texto”. Ésta es una relación de tipo “tiene un”, lo cual denota composición y no herencia. La clase que representa el formulario simplemente declara y crea objetos `Button` y `TextBox`, y después los utiliza.

Ahora regresemos al programa del juego para tratar de encontrar relaciones de herencia. Si lo conseguimos podremos simplificar y acortar el programa mediante la reutilización. En este caso varias de las clases (`Usuario`, `Extraterrestre`, `RayoLáser` y `Bomba`) incorporan las mismas propiedades: `x`, `y`, `Altura` y `Ancho`, que representan la posición y el tamaño de los objetos gráficos. A continuación eliminaremos estos ingredientes de cada clase para colocarlos en una superclase a la que denominaremos `Sprite`, término que se utiliza con frecuencia para describir objetos gráficos móviles en la programación de juegos. El diagrama UML para la clase `Sprite` es:

clase Sprite
Variables de instancia
coordenadaX
coordenadaY
altura
ancho
Propiedades
X
Y
Altura
Ancho

Este diseño se aclara al momento de analizar el código de la clase `Sprite`:

```
public class Sprite
{
    protected int xValor, yValor, anchoValor, alturaValor;

    public int X
    {
        get
        {
            return xValor;
        }
    }

    public int Y
    {
        get
        {
            return yValor;
        }
    }

    public int Ancho
    {
        get
        {
            return anchoValor;
        }
    }
}
```

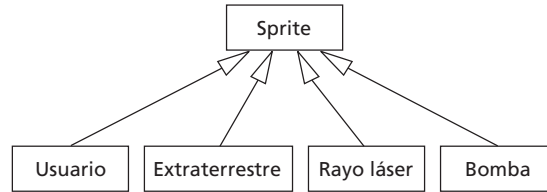


Figura 20.6 Diagrama de clases para los componentes heredados en el juego.

```

public int Altura
{
    get
    {
        return alturaValor;
    }
}

```

Las clases `Usuario`, `Extraterrestre`, `Rayo Láser` y `Bomba` ahora heredan estas propiedades de la superclase `Sprite`. Para comprobar la validez de este diseño decimos que “cada una de las clases, `Usuario`, `Extraterrestre`, `Rayo Láser` y `Bomba`, es un `Sprite`”. La Figura 20.6 muestra estas relaciones en un diagrama de clases. Recuerde que la flecha apunta de una subclase a una superclase.

Hemos podido identificar correctamente las relaciones de herencia entre las clases en el programa del juego.

A continuación nos enfocaremos en los detalles del programa. El código de la clase `Extraterrestre` es:

```

public class Extraterrestre : Sprite
{
    private int tamIntervalo;
    private PictureBox imagenExtraterrestre;

    public Extraterrestre(PictureBox imagenInicialExtraterrestre)
    {
        xValor = 0;
        yValor = 0;
        imagenExtraterrestre = imagenInicialExtraterrestre;
        anchoValor = imagenExtraterrestre.Ancho;
        alturaValor = imagenExtraterrestre.Altura;
        tamIntervalo = 1;
    }

    public void Dibujar(Graphics papel)
    {
        Image imagen = imagenExtraterrestre.Image;
        papel.DrawImage(imagen, xValor, yValor,
            anchoValor, alturaValor);
    }
}

```

```

public void Mover()
{
    if (xValor > 200 || xValor < 0)
    {
        tamIntervalo = -tamIntervalo;
    }
    xValor= xValor+ tamIntervalo;
}
}

```

El código para la clase Bomba es:

```

public class Bomba : Sprite
{
    private int tamIntervalo;
    public Bomba(int xInicial, int yInicial)
    {
        xValor = xInicial;
        yValor = yInicial;
        anchoValor = 5;
        alturaValor = 5;
        tamIntervalo = 1;
    }
    public void Dibujar(Graphics papel)
    {
        SolidBrush brocha = new SolidBrush(Color.Black);
        papel.FillEllipse(brocha, xValor, yValor,
                        anchoValor, alturaValor);
    }
    public void Mover()
    {
        yValor = yValor + tamIntervalo;
    }
}

```

## PRÁCTICA DE AUTOEVALUACIÓN

### 20.2 Escriba el código para la clase RayoLáser.

Para resumir, los dos tipos de relaciones entre clases son:

Relación entre clases	Prueba	El código de C# requiere
herencia	"es un"	:
composición	"tiene un" o "consiste en"	new

**PRÁCTICA DE AUTOEVALUACIÓN**

**20.3** Analice las relaciones que existen entre los siguientes grupos de clases (¿son del tipo “es un” o “tiene un”?):

1. casa, puerta, techo, vivienda
2. persona, hombre, mujer
3. automóvil, pistón, caja de cambios, motor
4. vehículo, automóvil, autobús

**● Lineamientos para el diseño de clases**

El uso de la metodología de diseño que hemos descrito no nos garantiza obtener un diseño perfecto. Siempre es conveniente valorar el diseño de cada clase de acuerdo con los siguientes lineamientos.

**Mantenga los datos privados**

Las variables siempre deben declararse como `private` (y algunas veces como `protected`), pero nunca como `public`. Esto mantiene el ocultamiento de datos, uno de los principios centrales de la programación orientada a objetos. Si necesitamos acceder a los datos o modificarlos, tendremos que hacerlo mediante propiedades o métodos que se proporcionan como parte de la clase.

**Inicialice los datos**

Aunque C# inicializa de manera automática las variables de instancia (pero no las variables locales) con valores específicos, es una buena práctica inicializarlas explícitamente, ya sea dentro de la misma declaración de datos o mediante un método constructor.

**Evite el uso de clases extensas**

Si una clase tiene más de dos páginas de texto, considere dividirla en dos o más clases de menor tamaño. No obstante, sólo debemos hacer esto si podemos identificar claramente las clases que se puedan formar a partir de la clase más grande; es contraproducente dividir una clase cohesiva y funcional en clases artificiales y poco elegantes.

**Dé nombres representativos a las clases, propiedades y métodos**

Esto hará que sean más fáciles de usar, y más atractivos para la reutilización.

**No invente la herencia**

En el programa de juegos anterior podríamos haber creado una superclase llamada `SeMueveHorizontalmente`, y hacer que `Usuario` y `Extraterrestre` fueran subclases de la misma. De manera similar, una clase llamada `SeMueveVerticalmente` podría haber sido la superclase de `RayoLáser` y `Bomba`.



Pero si consideramos los requerimientos individuales de estas clases, descubriremos que son bastante distintas y que no hubiéramos ganado nada al hacerlo.

Usar la herencia cuando no es realmente apropiada puede llevarnos a obtener clases artificiales, que son más complejas y tal vez más extensas de lo necesario.

### Al utilizar la herencia, coloque los elementos compartidos en la superclase

En el ejemplo de la cuenta bancaria que vimos antes, todas las variables, métodos y propiedades comunes para todas las cuentas bancarias deben colocarse en la superclase, para que todas las subclases puedan compartirlas sin tener que duplicarlas. Algunos ejemplos de esto son los métodos para actualizar la dirección del cuentahabiente y para actualizar su saldo.

También vimos en el programa Invasor del ciberespacio que podemos identificar propiedades idénticas en varias de las clases y, por ende, crear una superclase llamada **Sprite**.

### Use la refactorización

Después de crear un diseño inicial, o cuando hemos realizado cierta parte de la codificación, un estudio del diseño podría revelar la posibilidad de simplificación. Para ello tal vez tengamos que cambiar algunas propiedades y métodos a otra clase, crear nuevas clases, o amalgamar clases existentes. A este proceso se le conoce como *refactorización*. De hecho, ya hemos cumplido uno de los lineamientos de la refactorización: colocar las propiedades y métodos compartidos en la superclase.

Por ejemplo, en el programa Invasor del ciberespacio existe la necesidad obvia de evaluar varias veces si los objetos chocan. Pero hay varios lugares alternativos en donde puede llevarse a cabo esta detección de colisiones. Según la implementación que realizamos antes, hay un método llamado **colisiona**, el cual forma parte de la clase **Juego**, que lleva a cabo la detección de colisiones. Pero una alternativa sería crear una clase distinta, llamada **DetecciónDeColisiones**, que provea un método estático **colisiona** para detectar estos eventos. Además, podríamos dispersar la detección de colisiones a lo largo de todo el programa, en las clases **RayoLáser** y **Bomba**.

La refactorización reconoce que a menudo resulta imposible crear un diseño inicial perfecto. En realidad el diseño suele evolucionar hacia una estructura óptima. Para ello tal vez sea necesario modificarlo una vez avanzada la codificación. En consecuencia, el desarrollo no se lleva a cabo en distintas etapas.

## Resumen

La tarea de diseño orientado a objetos consiste en identificar los objetos y clases apropiados. Los pasos de la metodología de diseño orientado a objetos que vimos en este capítulo son:

1. Estudiar la especificación, y aclararla si es necesario.
2. Derivar los objetos, propiedades y métodos de la especificación, de manera que el diseño actúe como un modelo de la aplicación. Los verbos son métodos y los sustantivos son objetos.

3. Generalizar los objetos en clases.
4. Comprobar la reutilización de clases de biblioteca y otras clases existentes mediante el uso de la composición y la herencia, según sea apropiado. Las pruebas “es un” y “tiene un” nos ayudan a comprobar si es adecuado emplear la herencia o la composición.
5. Use los lineamientos para refactorizar sus diseños.

## EJERCICIOS

**20.1** Complete el desarrollo del programa Invasor del ciberespacio.

**20.2** Mejore el programa Invasor del ciberespacio, de manera que el extraterrestre pueda lanzar varias bombas y el usuario sea capaz de disparar varios rayos láser al mismo tiempo.

**20.3** Un buen diseño se puede juzgar a partir de qué tan bien se acopla a las modificaciones o mejoras. Considere las siguientes mejoras al programa Invasor del ciberespacio. Evalúe los cambios que sean necesarios en el diseño y la codificación:

- (a) una fila de extraterrestres;
- (b) una línea de trincheras que protejan al usuario de las bombas;
- (c) que se despliegue el desempeño del jugador (por ejemplo, mediante un marcador).

**20.4** Un diseño alternativo para el programa Invasor del ciberespacio utiliza una clase llamada `AdministradorDePantalla`, la cual:

- (a) mantiene la información sobre todos los objetos que aparecen en pantalla;
- (b) invoca todos los objetos para que se desplieguen a sí mismos;
- (c) detecta las colisiones entre pares de objetos en la pantalla.

Rediseñe el programa, de manera que emplee dicha clase.

**20.5** Diseñe las clases para un programa con la siguiente especificación:

El programa debe actuar como una calculadora sencilla de escritorio (Figura 20.7) que opere con números enteros. Debe haber un cuadro de texto para representar la pantalla; además, la calculadora tiene botones para cada uno de los diez dígitos, del 0 al 9, un botón para sumar y uno para restar. También tiene un botón para borrar la pantalla, y un botón de igual (=) para obtener la respuesta de los cálculos.

Cuando se oprima el botón para borrar la pantalla, ésta deberá mostrar un cero y el total (oculto) se establecerá en cero.

Cuando se oprima el botón de un dígito, éste se agregará a la derecha de los dígitos que ya se encuentren en el cuadro de texto (en caso de haber alguno).

Cuando se oprima el botón +, el número en el cuadro de texto se sumará al total (o se restará, en caso de que se utilice el botón -).

Cuando se oprima el botón de igual, se desplegará el valor del total.



Figura 20.7 La calculadora de escritorio.

## SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

### 20.1

```
clase Bomba
```

```
Variables de instancia
```

```
coordenadaX
```

```
coordenadaY
```

```
altura
```

```
ancho
```

```
yIntervalo
```

```
Métodos
```

```
Bomba
```

```
Mover
```

```
Dibujar
```

```
Propiedades
```

```
X
```

```
Y
```

```
Altura
```

```
Ancho
```

```
20.2  public class RayoLáser : Sprite
      {
          private int tamIntervalo;

          public RayoLáser(int nuevaX, int nuevaY)
          {
              xValor = nuevaX;
              yValor = nuevaY;
              anchoValor = 5;
              alturaValor = 5;
              tamIntervalo = 1;
          }

          public void Dibujar(Graphics papel)
          {
              SolidBrush pincel = new SolidBrush(Color.Black);
              papel.FillEllipse(pincel, xValor, yValor,
                              anchoValor, alturaValor);
          }

          public void Mover()
          {
              yValor = yValor - tamIntervalo;
          }
      }
}
```

- 20.3**
1. tiene un
  2. es un
  3. tiene un
  4. es un

# Estilo de programación

## En este capítulo conoceremos:

- estructuras de un programa;
- comentarios;
- constantes;
- clases;
- instrucciones `if` anidadas y ciclos;
- condiciones complejas;
- documentación.

## ● Introducción

---

La programación es una actividad muy creativa y emocionante. A menudo los programadores quedan absortos en su trabajo, y consideran que sus programas son creaciones muy personales. De acuerdo con el estereotipo, el programador (hombre o mujer) usa pantalones de mezclilla y camiseta, bebe veinte tazas de café al día, y permanece despierto toda la noche, sólo por la diversión de programar.

Pero la vida real de los programadores suele ser muy distinta. Casi siempre el trabajo de programación se lleva a cabo en organizaciones comerciales, y prácticamente todos los programas son resultado de la colaboración entre varias personas. De hecho, muchas organizaciones tienen manuales de estándares que detallan cómo debe ser la apariencia de los programas.

No hay duda de que al escribir un programa podemos vernos tentados a considerarlo como una creación individual, pero en casi todos los casos los programas son leídos por distintas personas, como quien se hace cargo de su trabajo cuando usted recibe un ascenso o cambia a otro proyecto, o los encargados de evaluar el software, y las generaciones de programadores que velarán por él, corrigiendo errores y mejorándolo mucho tiempo después de que usted haya obtenido otro empleo. Por lo tanto, hacer que su programa sea fácil de leer es un ingrediente vital de la programación.

Otro aspecto que tiene que ver con un buen estilo de programación es la capacidad de reutilización. Un programa bien construido debe contener clases que puedan reutilizarse más tarde en otros programas.

Aunque la opinión de las personas en cuanto al estilo de programación suele diferir, en lo que siempre están de acuerdo es en que este estilo debe aplicarse de manera consistente en todo el programa. Si el estilo es inconsistente, el programa será difícil de leer (por no decir molesto). También surge la preocupación de que al programador original en realidad no le importaba el programa, y en consecuencia, éste puede estar mal. En este libro hemos utilizado un estilo consistente para la estructura de los programas.

A menos que usted sea un aficionado, es importante saber cómo producir programas que tengan un estilo adecuado. Nos hemos esforzado por escribir programas funcionales que sirvan como buenos ejemplos a lo largo del libro.

## ● Estructura del programa

---

El programador en C# cuenta con muchas facilidades para decidir cómo estructurar un programa. El lenguaje es de formato libre: se pueden usar líneas y espacios en blanco casi en cualquier parte. Además, permite la colocación de comentarios en una o varias líneas, o al final de una línea de código. Hay mucho espacio para la creatividad e individualidad. Sin embargo, como hemos visto, casi todos los programas son leídos por varias personas, además del autor original. Por lo tanto, es imprescindible que el programa tenga una buena apariencia. A continuación analizaremos un conjunto de lineamientos de estilo para los programas creados con C#. Siempre hay controversia en cuanto a las directrices de este tipo, así que sin duda el lector estará en desacuerdo con algunas de las que se comentarán a continuación.

### Nombres

El programador asigna nombres a las variables, clases, propiedades y métodos. Hay mucho espacio para la imaginación, ya que los nombres pueden tener hasta 255 caracteres de longitud, siempre y cuando estén formados por letras, dígitos y guiones bajos; además, deben empezar con una letra.

Nuestro consejo en cuanto a los nombres es hacerlos lo más representativos que sea posible. En consecuencia, quedan descartados nombres enigmáticos como `i`, `j`, `x` y `y`, que por lo general indican que el programador tiene cierta experiencia con las matemáticas (pero no mucha imaginación para crear nombres representativos). A pesar de lo anterior, algunas veces los nombres cortos pueden ser apropiados. Por ejemplo, en este libro utilizamos muchas veces `x` y `y` para describir las coordenadas de un punto en un cuadro de imagen.

Es muy probable que llegue a verse en la necesidad de crear un nombre a partir de varias palabras. En esos casos, use letras mayúsculas para diferenciar un cambio de palabra; por ejemplo, `deseoQueEstésAquí`.

Algunos nombres empiezan con mayúscula, pero otros deberían empezar con minúscula. La convención sugerida es la siguiente:

#### **Empiezan con mayúscula:**

- clase
- método
- propiedad

#### **Empiezan con minúscula:**

- palabra reservada
- parámetro
- variable local
- variable de instancia
- componente de GUI (control)

El IDE de C# da nombres predeterminados a los controles. Estos nombres están bien mientras haya sólo un control de cada tipo, pero cuando hay varios —por ejemplo, varios botones o etiquetas—, le sugerimos reemplazar el nombre predeterminado con uno distintivo, como `botónCalcular`, `cuadroDeTextoEdad`, `etiquetaResultado`.

## Sangría o indentación

La sangría enfatiza la estructura del programa. Aunque C# y su compilador no necesitan usar sangría, nosotros podemos comprender mejor un programa cuando se aplica sangría a las instrucciones de selección y a los ciclos. Por fortuna el entorno de desarrollo integrado de Microsoft C# aplica automáticamente sangría a los programas. Si su codificación se vuelve confusa, seleccione el texto que desee ordenar y:

- haga clic en **Editar**;
- haga clic en **Avanzado**;
- haga clic en **Dar formato a la selección**.

## Líneas en blanco

Es frecuente que se utilicen líneas en blanco dentro de una clase, para separar las declaraciones de variables de las declaraciones de métodos y propiedades, y para separar un método o una propiedad de otros. Si hay muchas declaraciones de variables también se pueden separar distintos bloques de datos mediante líneas en blanco.

## Clases y archivos

Puede guardar todas sus clases en un solo archivo, pero tal vez sea mejor colocarlas en distintos archivos para dar la máxima facilidad de reutilizarlas. El IDE le ayudará con este proceso de mantenimiento.

## ● Comentarios

Hay dos formas de colocar comentarios en programas de C#:

```
// éste es un comentario al final de la línea

/* éste es un comentario
   que abarca
   varias líneas */
```

Siempre hay una gran controversia en cuanto al uso de comentarios en los programas. Algunas personas afirman que “entre más sean, mejor”. Sin embargo, en ocasiones podemos encontrarnos con código como el siguiente:

```
// muestra el mensaje 'hola'
textBox1.Text = "hola";
```

en donde el comentario simplemente repite lo que el código implica y, por lo tanto, resulta superfluo.

Hay veces que el código se ve invadido por comentarios sofocantes que ayudan muy poco a comprender mejor su significado. Es como un árbol de Navidad atestado de guirnaldas, adornos y luces, al grado que no podemos verlo por tantas decoraciones. Y ése no es el único problema: ciertos estudios han demostrado que cuando hay muchos comentarios, el lector tiende a concentrarse en ellos e ignorar el código. En consecuencia, si el código es incorrecto así se quedará.

Hay quien argumenta que los comentarios son necesarios cuando el código es complejo o difícil de entender. Esto parece razonable hasta que nos preguntamos por qué el código tendría que ser complejo en primer lugar. Tal vez el código se pueda simplificar de manera que sea fácil de comprender sin comentarios. A continuación le mostraremos ejemplos de dichas situaciones.

Algunos programadores prefieren colocar un comentario al inicio de cada clase y tal vez al inicio de una propiedad o método, para poder describir su propósito general. No obstante, como ya hemos comentado, los nombres de las clases, propiedades y métodos tratan de describir lo que hacen, por lo que un comentario podría ser redundante.

En cualquier caso, lo recomendable es utilizar los comentarios con medida. Por ejemplo, una sección compleja de código podría necesitar de un comentario explicativo.

## ● Uso de constantes

---

Muchos programas tienen valores que no cambian durante su ejecución, y en general se modifican con poca frecuencia. Algunos ejemplos son una tasa de impuestos, la edad necesaria para votar, el límite para pago de contribuciones y las constantes matemáticas. C# nos da la facilidad de declarar elementos de datos (como constantes) y asignarles un valor. Entonces, si tomamos en cuenta los ejemplos antes mencionados, podemos escribir:

```
const double tasaImpuestos = 17.5;
const int edadVotar = 18;
const int límiteImpuestos = 5000;
```

Variables como éstas, con valores constantes, sólo pueden declararse en la parte superior de una clase y no como variables locales dentro de un método.

Las cadenas de texto también pueden recibir valores constantes (pero las matrices no):

```
const string nuestroPlaneta = "Tierra";
```

Un beneficio de usar valores `const` es que el compilador detectará cualquier intento (sin duda erróneo) de cambiar el valor de una constante. Por ejemplo, dada la anterior declaración de `edadVotar`, la siguiente línea de código:

```
edadVotar = 17;
```

provocará un mensaje de error.

Un beneficio todavía mayor del uso de constantes es que un programa, en vez de estar invadido de números nada significativos, puede contener variables (que sean constantes) con nombres claros y significativos. Esto mejora la claridad del programa, con todas las ventajas que conlleva.



Por ejemplo, suponga que necesitamos modificar un programa de impuestos para reflejar un cambio en la legislación. Nuestra tarea será una pesadilla si todos los límites y las tasas de contribución están integrados en el programa como números que aparecen cada vez que se requieren en diversas partes del mismo. Suponga que el límite de impuestos anterior es de \$5000. Podríamos usar un editor de texto para buscar todas las ocurrencias de 5000. El obediente editor nos indicará en dónde se encuentran, pero no podemos estar seguros de que ese número tenga el mismo significado en todas partes. ¿Qué tal si aparece el número 4999 en el programa? ¿Será el límite de impuestos menos 1, o tiene algún otro significado sin relación alguna? Desde luego, la respuesta es utilizar constantes con nombres descriptivos, y asegurarnos de tener cuidado al diferenciar los distintos elementos de datos.

También podemos utilizar constantes para especificar el tamaño de los arreglos empleadas en un programa, como en el siguiente ejemplo:

```
const int tamaño = 10;
```

y por consiguiente:

```
int[] miArreglo = new int[tamaño];
```

Por último, algunas personas prefieren el estilo en el que los nombres de las constantes se escriben en **MAYÚSCULAS**, haciéndolos todavía más distintivos.

## ● Clases

Las clases son importantes bloques de construcción en los programas orientados a objetos. El correcto diseño de las clases nos ayuda a asegurarnos de que el programa sea claro y comprensible. El capítulo 20, dedicado al análisis del diseño orientado a objetos (DOO), explica una metodología. El DOO trata de crear clases que correspondan a ideas en el problema que buscamos resolver; por lo general estas clases están presentes en la especificación del programa. Por lo tanto, en un buen diseño debemos ser capaces de reconocer las clases como un modelo de la especificación. Como producto derivado, el diseño debe reflejar la complejidad del problema, y nada más.

Las clases constituyen también la unidad que facilita la reutilización de los componentes de software. Son precisamente los elementos que heredamos o ampliamos. En consecuencia, es importante que tengan un estilo sólido. He aquí algunos lineamientos para su creación.

### Tamaño de la clase

Si la codificación de una clase ocupa, digamos, más de dos páginas, tal vez sea demasiado grande y compleja. Considere dividirla cuidadosamente en dos o más clases, de manera que pueda crear nuevas clases viables. Tenga cuidado, sin embargo, ya que es peligroso dividir una clase coherente en varias clases confusas. En una clase coherente todas sus partes contribuyen para obtener un mismo objetivo.

### Tamaño de los métodos

Podríamos enfrascarnos en largas y divertidas discusiones respecto de cuál es la longitud apropiada de los métodos.

Un punto de vista podría ser que la codificación de un método no debe exceder el tamaño de la pantalla o el de la página de un listado (digamos, unas cuarenta líneas de texto). De esa forma no tenemos que desplazarnos ni cambiar de página para analizar a detalle todo el método. En otras palabras, hay que cuidar que no sea tan grande como para perder la pista de algunas de sus partes.

Cualquier método cuya codificación sea superior a media página se considera un serio candidato a ser reestructurado en varios métodos más pequeños. Sin embargo, depende de lo que el método haga; tal vez realice una sola tarea cohesiva, y si tratamos de dividirlo podríamos ocasionar complicaciones en cuanto a los parámetros y alcance. No aplique a ciegas ninguna recomendación relacionada con la longitud de un componente.

## Encapsulamiento

La idea del diseño orientado a objetos es ocultar o encapsular los datos, de manera que cualquier interacción entre clases se realice mediante las propiedades y métodos en vez de acceder directamente a los datos. El correcto diseño de una clase tendrá un mínimo de variables `public`.

## Nombres de propiedades y métodos

Ya hemos resaltado la importancia de dar nombres significativos a los métodos y las propiedades. Cuando la única función de un método es obtener algún valor, digamos, el valor de un sueldo, la convención es llamarlo `getSueldo`. De manera similar, si se va a proveer un método para cambiar el valor de esa misma variable, el nombre convencional será `setSueldo`.

## Orden de los campos

Los campos son las variables, propiedades y métodos declarados dentro de una clase. ¿En qué orden deben aparecer? Hay que considerar tanto los campos `public` como los `private`. La convención común es escribirlos en el siguiente orden:

1. variables de instancia (`public` y `private`);
2. métodos `public`;
3. propiedades;
4. métodos `private`.

## ● Instrucciones `if` anidadas

---

Anidar significa escribir una instrucción dentro de otra; por ejemplo, una instrucción `if` dentro de otra instrucción `if`, o un ciclo `while` dentro de un ciclo `for` (de lo cual hablaremos más adelante). A veces los programas anidados son simples y claros, pero en general un alto nivel de anidamiento se considera un estilo de programación incorrecto, y es mejor evitarlo. De hecho, se pueden evitar las instrucciones anidadas complejas mediante la reestructuración del programa.

Considere el problema de encontrar el mayor de tres números. He aquí un programa inicial que utiliza anidamiento:

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

if (a > b)
{
    if (a > c)
    {
        mayor = a;
    }
    else
    {
        mayor = c;
    }
}
else
{
    if (b > c)
    {
        mayor = b;
    }
    else
    {
        mayor = c;
    }
}
```

Sin duda este programa se ve complicado, y tal vez a algunas personas se les dificulte un poco comprenderlo. La complejidad se debe al anidamiento de las instrucciones `if`.

El siguiente programa evita el anidamiento, pero aumenta la complejidad de las condiciones:

```
if (a >= b && a >= c)
{
    mayor = a;
}
if (b >= a && b >= c)
{
    mayor = b;
}
if (c >= a && c >= b)
{
    mayor = c;
}
```

Con todo, este programa puede ser más legible para ciertas personas.

Hemos examinado dos soluciones al mismo problema. En efecto, a menudo existe más de una solución para un problema, y cada una de ellas tiene sus propias fortalezas y debilidades.

Puede ser difícil leer y comprender programas que utilicen anidamiento. Nuestros ejemplos muestran cómo un programa en el que se utilizan instrucciones `if` anidadas puede convertirse en un programa sin anidamiento; en general esto puede lograrse con cualquier programa. Pero las instrucciones `if` anidadas no siempre son malas; hay ocasiones en que el anidamiento describe con simpleza y claridad la tarea a realizar.

## ● Ciclos anidados

---

Veamos ahora los ciclos anidados. Suponga que debemos escribir un programa que despliegue en pantalla un patrón como el de la Figura 21.1, el gráfico simple de un bloque de apartamentos.

El programa podría ser algo así:

```
private void DibujarApartamentos(int pisos, int apartamentos)
{
    int coordX, coordY;
    Graphics papel = pictureBox1.CreateGraphics();
    Pen lápiz = new Pen(Color.Black);

    coordY = 10;
    for (int piso = 1; piso <= pisos; piso++)
    {
        coordX = 10;
        for (int conteo = 1; conteo <= apartamentos; conteo++)
        {
```

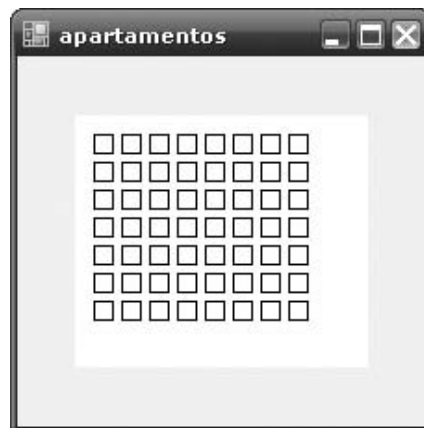


Figura 21.1 Despliegue del bloque de apartamentos.

```

        papel.DrawRectangle(lápiz, coordX, coordY, 10, 10);
        coordX = coordX + 15;
    }
    coordY = coordY + 15;
}
}

```

En este programa un ciclo está anidado dentro de otro. No es una pieza de código compleja por sí misma, pero de todas formas podemos simplificarla si utilizamos otro método llamado `DibujarPiso`:

```

private void DibujarApartamentos2(int pisos, int apartamentos)
{
    int coordY;

    coordY = 10;
    for ( int piso = 1; piso <= pisos; piso++)
    {
        DibujarPiso(coordY, apartamentos);
        coordY = coordY + 15;
    }
}

private void DibujarPiso(int coordY, int apartamentos)
{
    int coordX = 10;

    Graphics papel = pictureBox1.CreateGraphics();
    Pen lápiz = new Pen(Color.Black);

    for (int conteo = 1; conteo <= apartamentos; conteo++)
    {
        papel.DrawRectangle(lápiz, coordX, coordY, 10, 10);
        coordX = coordX + 15;
    }
}

```

Gracias al método adicional pudimos eliminar el anidamiento. Además, expresamos explícitamente en el código el hecho de que el bloque de apartamentos consiste en varios pisos. Hemos desmenuzado dos problemas en piezas separadas; algunas veces a esto se le conoce como descomposición del problema. También aclaramos el requerimiento de que hay un cambio en la coordenada *y* para cada piso del bloque. Siempre es posible eliminar ciclos anidados de esta forma, y en ocasiones esto produce una simplificación del programa.

Algunos estudios de investigación han mostrado que a los seres humanos se nos dificulta comprender programas que utilicen anidamiento. Un investigador resumió este hecho al decir: “El anidamiento es para las aves”. Pero *no siempre* es malo. Considere el ejemplo del código para inicializar una matriz bidimensional:

```
int [,] tabla = new int[9,9];

for (int fila = 0; fila <= 8; fila++)
{
    for (int col = 0; col <= 8; col++)
    {
        tabla[fila, col] = 0;
    }
}
```

Este ejemplo es claro, aunque use anidamiento.

## ● Condiciones complejas

---

La complejidad de una instrucción `if`, `for`, `while` o `do` puede surgir cuando la condición a evaluar implica uno o más operadores `&&` y/o `||`. Una condición compleja es capaz de provocar que un programa sea muy difícil de entender, depurar y corregir. Como ejemplo analizaremos un programa que busca el número deseado en un arreglo numérico:

```
const int tamaño = 100;
int[] tabla = new int[tamaño];

int buscado = Convert.ToInt32(buscadoTextBox.Text);

int índice;
for (índice = 0; índice < (tamaño - 1) && tabla[índice] != buscado; índice++)
{
}
if (tabla[índice] == buscado)
{
    ResultadoTextBox.Text = "se encontró";
}
else
{
    ResultadoTextBox.Text = "no se encontró";
}
```

El problema con este programa es que la condición incluida en el ciclo `for` es complicada (aunque el cuerpo del ciclo esté vacío). Incluso para un programador experimentado puede ser difícil comprobar lo que se escribió y convencerse de que es correcto. Hay una alternativa: utilizaremos una bandera. Se trata simplemente de una variable `int`, pero su valor en cualquier momento registra el estado de la búsqueda. Son tres los estados que pueden darse en relación con la búsqueda:

- El programa aún está buscando; no se ha encontrado el elemento. Éste es también el estado inicial de la búsqueda. La bandera tiene el valor de 0.

- El elemento se encontró. El valor de la bandera es 1.
- La búsqueda se completó sin encontrar el elemento. El valor de la bandera es 2.

Si utilizamos esta bandera, a la que llamaremos **estado**, el programa quedará así:

```
const int tamaño = 100;
int[] tabla = new int[tamaño];

int estado;
const int sigueBuscando = 0;
const int encontró = 1;
const int noEncontró = 2;

int buscado = Convert.ToInt32(entradaTextBox.Text);

estado = sigueBuscando;

for (int índice = 0; estado == sigueBuscando; índice++)
{
    if (buscado == tabla[índice])
    {
        estado = encontró;
    }
    else
    {
        if (índice == (tamaño - 1))
        {
            estado = noEncontró;
        }
    }
}

if (estado == encontró)
{
    resultadoTextBox.Text = "se encontró";
}
else
{
    resultadoTextBox.Text = "no se encontró";
}
```

Lo que logramos aquí es desenmarañar las diversas pruebas. La condición del ciclo `for` es clara y directa. Las demás pruebas son separadas y simples. El programa en general es tal vez más sencillo. En el capítulo 14 se presentó otra forma de escribir un programa de búsqueda con arreglos unidimensionales.

La moraleja es que muchas veces es posible escribir una pieza de programa de distintas maneras, y que algunas soluciones son más simples y claras que otras. En otras ocasiones es posible evitar la complejidad de una condición al reestructurar el fragmento del programa mediante el uso de una bandera.

## ● Documentación

---

La documentación es la pesadilla del programador... ¡hasta que se le pide que explique el funcionamiento del programa de otra persona! Por lo general las organizaciones comerciales tratan de alentar a los programadores para que documenten bien sus programas. Les cuentan la antigua y tal vez imaginaria historia sobre el programador que tenía listo noventa y cinco por ciento de un programa pero no había hecho documentación alguna... y más tarde lo atropelló un autobús. Supuestamente los colegas que se hicieron cargo de su trabajo tuvieron muchas dificultades para continuar trabajando en el programa.

Por lo común la documentación de un programa consiste en los siguientes ingredientes:

- la especificación del programa;
- impresiones de pantalla;
- el código fuente, incluyendo los comentarios apropiados;
- información de diseño, por ejemplo, los diagramas de clases;
- el plan de pruebas;
- los resultados de las pruebas;
- el historial de modificaciones;
- el manual del usuario.

Si alguna vez le piden hacerse cargo del programa creado por otra persona, eso es lo que necesitará... ¡pero no espere obtenerlo!

En general, los programadores consideran que la creación de documentación es una tarea aburrida y tienden a mostrarse negligentes al respecto. Es común que la dejen hasta el final del proyecto, cuando hay poco tiempo disponible. Así las cosas, no es de sorprender que casi siempre la documentación resulte muy deficiente, o sencillamente inexistente.

La única forma de aliviar el dolor consiste en realizar la documentación a medida que vamos avanzando, mezclándola con las tareas más interesantes de la programación.

## Errores comunes de programación

Compruebe si hay estándares que utilice su empresa antes de empezar a codificar. Tal vez tenga que seguirlos. Si desea apegarse a un plan para elaborar la estructura del programa, será mejor hacerlo desde el principio en vez de apresurarse a escribir el código del programa y modificarlo después.

## Resumen

- El estilo de un programa es importante, ya que mejora su legibilidad para los procesos de depuración y mantenimiento.
- Algunos lineamientos para mantener una buena estructura de los programas son: el uso de nombres apropiados, la utilización de sangría y líneas en blanco, y la adición de comentarios.
- C# cuenta con una herramienta útil para que los elementos de datos apropiados sean constantes.
- Las clases deben tener un propósito cohesivo claro.



- Las instrucciones `if`, los ciclos y las condiciones complejas anidadas deben utilizarse con medida.
- Vale la pena acompañar nuestros programas con una buena documentación.

## EJERCICIOS

- 21.1** En un programa para jugar cartas, el palo de cada una se codifica como un entero (del 1 al 4). Considere un método para convertir ese entero en la cadena apropiada: "corazones", "tréboles", etc. Escriba el método de cuatro maneras:
- (a) use instrucciones `if` anidadas;
  - (b) use instrucciones `if` distintas;
  - (c) use `switch case`;
  - (d) use un arreglo de cadenas de caracteres que contenga las cuatro cadenas.
- ¿Cuál solución es mejor, y por qué?
- 21.2** Analice todos los programas que pueda (incluyendo los suyos) y revise sus estilos. ¿Son correctos o incorrectos? ¿Por qué?
- 21.3** Discuta la cuestión de los lineamientos con sus colegas o amigos. ¿Es importante el estilo? De ser así, ¿qué es lo que constituye un buen estilo?
- 21.4** Desarrolle un conjunto de lineamientos de estilo para los programas de C#.
- 21.5** (Opcional) Use sus lineamientos de estilo para siempre.

# La fase de pruebas

## En este capítulo conoceremos:

- por qué no es factible realizar una prueba exhaustiva;
- cómo llevar a cabo una prueba funcional;
- cómo efectuar una prueba estructural;
- cómo realizar recorridos;
- cómo llevar a cabo el proceso de prueba mediante el recorrido paso a paso por las instrucciones del programa;
- la función de la verificación formal;
- el desarrollo incremental.

## ● Introducción

---

Los programas son complejos y es difícil lograr que funcionen correctamente. La fase de pruebas es el conjunto de técnicas que se utilizan para tratar de verificar que un programa trabaje de manera correcta. Dicho de otra forma, el proceso de prueba trata de revelar la existencia de errores. Una vez que detectamos un error, necesitamos localizarlo mediante el uso de la depuración (vea el capítulo 9). Como veremos más adelante, las técnicas de prueba no pueden garantizar la exposición de todos los errores de un programa y, por lo tanto, casi todos los programas grandes tienen fallas ocultas. Sin embargo, el proceso de prueba es muy importante, demostrado por el hecho de que por lo general consume hasta la mitad del tiempo total invertido en el desarrollo de un programa. Por ejemplo, en Microsoft hay tantos equipos de desarrolladores (quienes escriben los programas) como equipos de evaluadores (quienes los someten a prueba). Como se requiere mucho tiempo y esfuerzo para probar y depurar los programas, a menudo es preciso decidir si debemos continuar las pruebas o entregar el programa en su estado actual a los clientes.

En los círculos académicos, a la tarea de tratar de asegurar que un programa haga lo esperado se le conoce como *verificación*. El objetivo es *verificar* que un programa cumpla su especificación.

En este capítulo veremos cómo llevar a cabo el proceso de prueba de manera sistemática, revisaremos distintas metodologías de verificación, y veremos cuáles son sus deficiencias.

Las técnicas que revisaremos son:

- caja negra o prueba funcional;
- caja blanca o prueba estructural;
- revisiones o recorridos;
- recorrer paso a paso las instrucciones del código con un depurador;
- métodos formales.

Por lo general los programas pequeños, que sólo constan de una clase, pueden probarse de una sola vez. En cambio, los programas de mayor tamaño, digamos con dos o más clases, podrían llegar a tener una complejidad tal que sea conveniente probarlos clase por clase. A esto se le conoce como *prueba de unidad*. A la tarea de reunir el programa completo para evaluarlo se le conoce como prueba de *integración* o *de sistema*.

También veremos cómo desarrollar un programa parte por parte, en vez de desarrollarlo completo.

## ● Especificaciones de los programas

El punto inicial para cualquier prueba es la especificación. Nunca se dedica demasiado tiempo a estudiar y aclarar la especificación. Tal vez incluso sea necesario pedir algunas aclaraciones al cliente o al futuro usuario del programa. Por ejemplo, veamos la siguiente especificación:

Escriba un programa que reciba como entrada una serie de números mediante un cuadro de texto. El programa debe calcular y desplegar la suma de los números.

Al leerla por primera vez, esta especificación podría parecer simple y clara. Pero, aun siendo tan corta, contiene algunas omisiones:

- ¿Los números deben ser enteros o de punto flotante?
- ¿Cuáles son el rango y la precisión permisibles de los números?
- ¿La suma debe incluir números negativos?

Debemos aclarar estas dudas antes de que el programador empiece a trabajar. De hecho, parte del trabajo del programador es Analizar la especificación, descubrir si hay omisiones o confusiones, y hacer acuerdos para tener una especificación lo más clara posible,. Después de todo, no tiene caso escribir un programa brillante si no hace lo que el cliente esperaba.

Veamos una versión más clara de la especificación, misma que utilizaremos como ejemplo práctico para analizar los métodos de prueba:

Escriba un programa que reciba como entrada una serie de números enteros mediante un cuadro de texto. Los enteros deben estar en el rango de 0 a 10,000. El programa debe calcular y desplegar la suma de los números.

Como podemos ver, esta especificación es más precisa; por ejemplo, estipula el rango permisible de los valores de entrada.

**PRÁCTICA DE AUTOEVALUACIÓN****22.1 ¿Detecta alguna otra deficiencia en la especificación que debamos aclarar?****● Prueba exhaustiva**

Una metodología para el proceso de prueba consistiría en evaluar un programa con todos los valores de datos posibles como entrada. Pero consideremos incluso al más simple de los programas que reciba como entrada un par de números enteros y muestre su producto. La prueba exhaustiva implicaría seleccionar todos los valores posibles para el primer número y todos los valores posibles para el segundo y luego utilizar todas las posibles combinaciones de estos números. En C# un número `int` tiene un enorme rango de valores posibles; la cantidad de probables combinaciones de números es enorme. Tendríamos que introducir mediante el teclado todos los distintos valores, y ejecutar el programa. El tiempo requerido para ensamblar los datos de prueba sería de años; incluso el tiempo requerido por las computadoras sería de días, y eso que son rápidas. Por último, el proceso de comprobar que la computadora haya obtenido las respuestas correctas volvería loco de aburrimiento a cualquiera.

Debido a todo lo anterior, no es factible realizar una prueba exhaustiva, ni siquiera en el caso de un programa pequeño y simple. Es importante reconocer la imposibilidad de realizar un proceso de prueba perfecto para todos los programas, excepto los más pequeños. Por lo tanto, debemos adoptar alguna otra metodología.

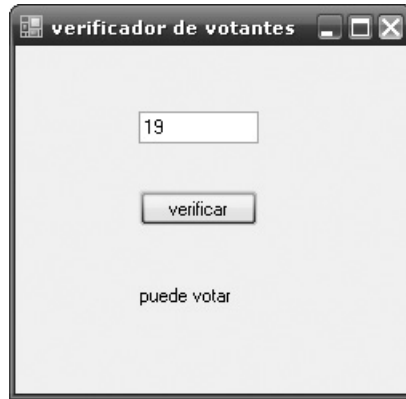
**● Prueba de la caja negra (funcional)**

Sabiendo que realizar una prueba exhaustiva resulta prácticamente imposible, podemos analizar una metodología que consiste en idear datos de muestra que representen todos los datos posibles. Después ejecutamos el programa, introducimos los datos y vemos qué ocurre. A este tipo de prueba se le denomina prueba de la *caja negra* debido a que no necesitamos conocer el funcionamiento del programa para realizar las pruebas; sólo consideramos las entradas y salidas. Partimos de la suposición de que el programa está completamente encerrado dentro de una caja negra invisible. A la prueba de la caja negra también se le conoce como prueba funcional, ya que sólo hay que conocer la función que desempeña el programa (y no cómo trabaja).

Lo ideal es anotar los datos de prueba y los resultados esperados antes de realizar la prueba. A esto se le conoce como especificación o programación de la prueba. Después ejecutamos el programa, introducimos los datos y examinamos los resultados para ver si hay discrepancias entre el resultado esperado y el resultado real. Los datos de prueba también deben verificar que el programa maneja las excepciones de acuerdo con su especificación.

Consideremos un programa que decide si una persona puede votar dependiendo de su edad (Figura 22.1). La edad mínima para votar son los 18 años.

Sabemos que no podemos probar este programa con todos los valores posibles, así que lo haremos con algunos valores típicos. La metodología de idear datos para la prueba de la caja negra consiste en utilizar una *partición equivalente*. Esto implica analizar la naturaleza de los datos de entrada



**Figura 22.1** El programa Verificador de votantes.

para identificar características comunes, a las que se les denomina partición. En el programa para votar reconocemos que hay dos particiones para los datos de entrada:

- los números menores que 18;
- los números mayores o iguales que 18.

Podemos hacer un diagrama de esto, como se muestra a continuación:

0	17	18	infinito
---	----	----	----------

Hay dos particiones, una incluye el rango de edades de 0 a 17, y la otra los números del 18 al infinito. Para los objetivos de prueba de este programa, el siguiente paso es asegurarnos de poder afirmar que todos los números dentro de una partición son equivalentes a cualquier otro (por ello se les llama partición equivalente). En otros términos, debemos argumentar que el número 12 es equivalente a cualquier otro en la primera partición, y que el número 21 es equivalente a cualquier otro en la segunda partición. Para ello desarrollamos dos pruebas:

Número de prueba	Datos	Resultado
1	12	No puede votar
2	21	Puede votar

Nuestro razonamiento podría ser que necesitamos dos conjuntos de datos de prueba para evaluar este programa. Estos dos conjuntos, aunados a una declaración de los resultados esperados, constituyen una especificación de prueba. Ejecutamos el programa con los dos conjuntos de datos, y observamos cualquier discrepancia entre los resultados esperados y los reales.

Por desgracia, podemos ver que estas pruebas no han investigado la importante distinción entre alguien de 17 años y alguien de 18. Cualquiera que haya escrito un programa sabe que se pueden cometer errores al usar instrucciones `if`, por lo que es recomendable examinar esa región específica

de los datos. Esto es lo mismo que reconocer que vale la pena incluir en la prueba los valores de los datos en los extremos de las particiones. Por ende, creamos dos pruebas adicionales:

Número de prueba	Datos	Resultado
3	17	No puede votar
4	18	Puede votar

En resumen, las reglas para seleccionar los datos para la prueba de la caja negra mediante el uso de la partición equivalente son:

1. Crear una partición de los valores de los datos de entrada.
2. Seleccionar datos representativos de cada partición (datos equivalentes).
3. Seleccionar los datos que se ubican en los límites de las particiones.

Ya hemos seleccionado los datos para la prueba. En este programa hay una sola entrada; existen cuatro valores de datos y, por lo tanto, cuatro pruebas. Sin embargo, casi todos los programas procesan varias entradas. Suponga que deseamos probar un programa que muestra el mayor de dos números, cada uno en el rango de 0 a 10,000. Los números se introducen en un par de cuadros de texto; si los valores son iguales, el programa muestra cualquiera de los dos.

Cada entrada está dentro de una partición que contiene los valores del 0 al 10,000. Elegimos los valores que están en cada extremo de las particiones, y valores de muestra ubicados en algún punto intermedio:

Primer número:	0	54	10000
Segundo número:	0	142	10000

Una vez seleccionados los valores representativos, necesitamos considerar cuáles combinaciones de valores debemos usar. La prueba exhaustiva implica utilizar cada posible combinación de todos los posibles valores de datos, lo cual, desde luego, es imposible. En vez de ello utilizamos todas las combinaciones de los valores representativos. Entonces, las pruebas son:

Número de prueba	1er. Número	2do. Número	Resultado
1	0	0	0
2	0	142	142
3	0	10000	10000
4	54	0	54
5	54	142	142
6	54	10000	10000
7	10000	0	10000
8	10000	142	10000
9	10000	10000	10000

De esta forma, el paso adicional en la prueba consiste en utilizar todas las combinaciones de los valores de datos representativos (limitados).

### PRÁCTICA DE AUTOEVALUACIÓN

**22.2** En un programa para jugar ajedrez, el jugador especifica el destino de un movimiento como un par de índices: los números de fila y de columna. El programa comprueba que la casilla de destino sea válida; es decir, que no esté fuera del tablero. Desarrolle los datos de una prueba de caja negra para comprobar que esta parte del programa funcione correctamente.

## ● Prueba de la caja blanca (estructural)

En la prueba de la caja blanca debemos saber cómo funciona el programa (su estructura) como base para desarrollar los datos de prueba. Durante el proceso de prueba de caja blanca cada instrucción del programa se ejecuta en cierto momento. Esto equivale a asegurar que se ejecutarán todas las rutas (todas las secuencias de instrucciones) del programa en algún momento durante el periodo de prueba. Se incluyen las rutas nulas, por lo que una instrucción `if` sin `else` tiene dos rutas, y cada ciclo tiene dos rutas. El proceso de prueba también debe incluir el manejo de excepciones que el programa lleve a cabo.

El siguiente es el código del programa verificador de votantes que utilizamos como ejemplo práctico:

```
private void button1_Click(object sender, EventArgs e)
{
    int edad;
    edad = Convert.ToInt32(textBox1.Text);
    if (edad >= 18)
    {
        label1.Text = "puede votar";
    }
    else
    {
        label1.Text = "no puede votar";
    }
}
```

En este programa hay dos rutas (debido a que la instrucción `if` tiene dos ramificaciones); por lo tanto, bastará con dos conjuntos de datos para asegurar que se ejecuten todas las instrucciones en un momento dado durante la fase de pruebas:

Número de prueba	Datos	Resultado esperado
1	12	No puede votar
2	21	Puede votar

Si somos cuidadosos tendremos en cuenta que a menudo los errores en la programación se cometen en las condiciones de las instrucciones `if` y `while`. Por ende, debemos agregar dos pruebas que nos permitan garantizar que la instrucción `if` funcione correctamente:

Número de prueba	Datos	Resultado esperado
3	17	No puede votar
4	18	Puede votar

Entonces necesitamos cuatro conjuntos de datos para probar este programa con el método de la caja blanca. En este caso son los mismos datos que desarrollamos para la prueba de la caja negra, pero el razonamiento que condujo a cada conjunto es distinto. Si el programa estuviera escrito de manera diferente, los datos para la prueba de caja blanca serían diferentes. Por ejemplo, suponga que el programa utiliza un arreglo llamado `tabla` con un elemento para cada edad, que especifica si alguien de esa edad puede votar o no. En tales circunstancias el programa sólo consistirá en la siguiente instrucción para ver si el candidato puede votar:

```
label1.Text = tabla[ edad ];
```

y los datos de prueba de la caja blanca serán distintos.

## PRÁCTICAS DE AUTOEVALUACIÓN

**22.3** La función de un programa es encontrar el mayor de tres números. Desarrolle datos para realizar una prueba de caja blanca para esta sección del programa.

El código es:

```
int a, b, c;
int mayor;
if (a >= b)
{
    if (a >= c)
    {
        mayor = a;
    }
    else
    {
        mayor = c;
    }
}
else
{
    if (b >= c)
    {
        mayor = b;
    }
}
```



```
    else
    {
        mayor = c;
    }
}
```

**22.4** En un programa para jugar ajedrez el jugador especifica el destino de un movimiento como un par de subíndices enteros: los números de fila y de columna. El programa verifica que la casilla de destino sea válida; es decir, que no esté fuera del tablero. Desarrolle datos para realizar una prueba de la caja blanca y comprobar que esta parte del programa funcione correctamente.

El código para esta parte del programa es:

```
if ((fila > 8) || (fila < 1))
{
    MessageBox.Show("error");
}
if ((col > 8) || (col < 1))
{
    MessageBox.Show("error");
}
```

## ● Inspecciones y recorridos

En la metodología de inspección o recorrido no utilizamos la computadora para tratar de erradicar los errores. En este caso la prueba consiste en una inspección visual del listado del programa (junto con la especificación), para tratar de detectar los errores. La inspección funciona mejor si la persona que la realiza no es quien escribió el programa. Esto se debe a que los seres humanos tendemos a cegarnos respecto de nuestros propios errores. Lo mejor, por lo tanto, es pedir a un amigo o colega que inspeccione su programa. Es extraordinario presenciar lo rápido que otra persona puede detectar un error que nos ha estado eludiendo durante horas.

Para inspeccionar un programa necesitamos:

- la especificación;
- una impresión del texto del programa.

Una de las metodologías para llevar a cabo una inspección consiste en estudiar un método a la vez. Algunas de las comprobaciones son bastante mecánicas:

- que las variables estén inicializadas;
- que los ciclos se inicien y terminen de manera correcta;
- que las invocaciones a métodos cuenten con los parámetros correctos.

Una comprobación más a fondo examina la lógica del programa. Pretenda ejecutar el método como si usted fuera una computadora, evitando seguir las invocaciones de un método a otro (ésta es la razón por la que esta metodología es conocida como recorrido). Debe comprobar que:

- la lógica del método obtenga el propósito deseado.

Durante la inspección puede comprobar que:

- los nombres de los métodos y variables sean representativos;
- la lógica sea clara y correcta.

Aunque el objetivo principal de una inspección no sea comprobar el estilo, una debilidad en cualquiera de estas áreas pudiera indicarnos la presencia de un error.

La evidencia obtenida de los experimentos controlados nos sugiere que las inspecciones constituyen una forma muy efectiva de encontrar errores. De hecho las inspecciones son por lo menos tan efectivas para identificar errores como las pruebas en las que tenemos que ejecutar el programa.

### ● **Recorrer paso a paso las instrucciones del programa**

---

El depurador incluido en el IDE de C# (capítulo 9) le permite avanzar paso a paso por un programa, ejecutando sólo una instrucción a la vez. A esto se le conoce como recorrido paso a paso por las instrucciones del programa. Cada vez que ejecutamos una instrucción podemos ver la ruta de ejecución que ha tomado el programa, y también los valores de las variables, al colocar el cursor sobre el nombre de una de ellas. Es algo así como un recorrido estructurado automatizado.

En este tipo de prueba nos concentramos en las variables, y comprobamos minuciosamente sus valores a medida que el programa las va modificando, para verificar que sus valores hayan cambiado correctamente.

Mientras el depurador por lo general se utiliza para la localización de errores, esta técnica se emplea para realizar pruebas, es decir, para negar o confirmar la existencia de un error.

### ● **Verificación formal**

---

Los métodos formales aprovechan la precisión y el poder de las matemáticas para tratar de verificar que un programa cumpla con su especificación. Después se concentran en la precisión de la especificación, la cual debe volverse a escribir en una notación matemática formal. Uno de los lenguajes de especificación utilizados es Z. Una vez escrita la especificación formal para un programa, existen dos metodologías alternativas:

1. Escribir el programa y luego verificar que se acople a la especificación. Para ello se requiere mucho tiempo y experiencia.
2. Derivar el programa de la especificación mediante una serie de transformaciones, cada una de las cuales se encarga de preservar la corrección del producto. En la actualidad ésta es la metodología preferida.

La verificación formal es muy atractiva, debido a su potencial para verificar con rigor la corrección de un programa, más allá de toda posible duda. Sin embargo, debemos recordar que estos métodos son llevados a cabo por seres humanos, susceptibles de cometer errores. Por lo tanto, no son infalibles.

La verificación formal aún se encuentra en su infancia y, en consecuencia, no se utiliza mucho en la industria o el comercio, salvo en unas cuantas aplicaciones de seguridad críticas. Debido a que esta metodología se encuentra más allá del alcance de este libro, nuestro análisis concluye en este punto.

## ● Desarrollo incremental

Una de las metodologías utilizadas para escribir un programa es escribir su codificación completa en papel, introducirla mediante el teclado y tratar de ejecutarla. La palabra importante en este caso es “tratar”, ya que casi todos los programadores descubren que el amistoso compilador detecta muchos errores en su programa. Puede ser muy desalentador (en especial para los principiantes) ver tal despliegue de errores en un programa cuya escritura implicó tanto esfuerzo. Una vez desterrados los errores de compilación, el programa por lo general exhibirá comportamientos extraños durante el, a veces extenso, periodo de depuración y prueba. Si todas las partes de un programa se introducen a la vez mediante el teclado para realizar la fase de pruebas, puede ser difícil localizar los errores. Una técnica útil para ayudarnos a evitar la frustración derivada consiste en hacer las cosas parte por parte. Así, una alternativa a desarrollar todo a la vez es la programación por bloques, tarea que suele denominarse programación *incremental*. Los pasos son:

1. Diseñar y construir la interfaz de usuario (el formulario).
2. Escribir una pequeña parte del programa.
3. Introducirla mediante el teclado, corregir los errores de sintaxis, ejecutarla y depurarla.
4. Agregar una parte nueva y pequeña del programa.
5. Repetir desde el paso 2 hasta que el programa esté completo.

El truco es identificar la parte del programa con la que podemos empezar y en qué orden debemos realizar las cosas. Tal vez la mejor metodología sea empezar por escribir el más simple de los métodos manejadores de eventos, y después los métodos que son utilizados por ese primer método. Más tarde podemos escribir otro método manejador de eventos, y así sucesivamente.

## Fundamentos de programación

No hay un método de prueba infalible que pueda asegurarnos que un programa esté libre de errores. La mejor metodología sería utilizar una combinación de los métodos de prueba —caja negra, caja blanca e inspección—, ya que la experiencia nos ha demostrado que encuentran distintos errores. Sin embargo, se requiere mucho tiempo para utilizar los tres métodos. En consecuencia, hay que ser muy precavidos y habilidosos para decidir qué tipo de prueba realizar y por cuánto tiempo se debe llevar a cabo. Asimismo, es vital contar con una metodología sistemática.

Con la prueba incremental evitamos actuar como si buscáramos una aguja en un pajar, ya que es probable que un error recién descubierto se encuentre en la clase que acabamos de incorporar.

El proceso de prueba es algo frustrante; porque sabemos que sin importar cuán pacientes y sistemáticos seamos, nunca podremos estar seguros de haber hecho lo suficiente. Para realizar las pruebas se requiere mucha paciencia, atención a los detalles y organización.

Escribir un programa es una experiencia constructiva, pero realizar pruebas es un proceso destructivo. Puede ser difícil tratar de demoler un objeto que nos ha tomado horas crear, y del cual nos sentimos orgullosos; al encontrar un error necesitaremos más horas todavía para poder rectificar el problema. Por todo esto, es muy fácil comportarnos evasivos durante las pruebas, tratando de evitar los problemas.

## Resumen

- La prueba es una técnica que trata de establecer que un programa no contiene errores.
- La prueba no puede ser exhaustiva, debido a que hay demasiados casos que probar.
- La prueba de la caja negra sólo utiliza la especificación para elegir los datos de prueba.
- En la prueba de la caja blanca debemos conocer cómo funciona el programa para poder elegir los datos de prueba.
- La inspección simplemente implica estudiar la codificación del programa para ver si encontramos errores.
- Avanzar paso a paso por cada instrucción del código mediante un depurador puede ser una valiosa manera de probar un programa.
- El desarrollo incremental puede evitar las complejidades que surgen al desarrollar programas extensos.

## EJERCICIOS

**22.1** Invente datos para realizar las pruebas de la caja negra y de la caja blanca en el siguiente programa. La especificación es:

*El programa recibe como entrada números enteros mediante el uso de un cuadro de texto y un botón. El programa muestra el mayor de los números introducidos hasta un momento dado.*

Trate de no analizar el texto del programa que le mostraremos a continuación antes de haber completado el diseño de los datos para la prueba de la caja negra.

A nivel de clase tenemos la declaración de una variable de instancia:

```
private int mayor = 0;
```

El código para manejar eventos es:

```
private void button1_Click(object sender, EventArgs e)
{
    int número;
    número = Convert.ToInt32(textBox1.Text);
    if (número > mayor)
    {
        mayor = número;
    }
    label1.Text = "el número más grande hasta el momento es " +
    Convert.ToString(mayor);
}
```

**22.2** Invente datos para realizar las pruebas de la caja negra y de la caja blanca en el siguiente programa. La especificación se muestra a continuación. Trate de no analizar el texto del programa que le mostramos antes de haber completado el diseño de los datos para la prueba de la caja negra.

*El programa determina las primas de seguros para un día festivo, con base en la edad y el género (masculino o femenino) del cliente.*

*Para una mujer cuya edad sea  $\geq 18$  y  $\leq 30$ , la prima es de \$5.*

*Para una mujer cuya edad sea  $\geq 31$ , es de \$3.50.*

*Para un hombre cuya edad sea  $\geq 18$  y  $\leq 35$ , es de \$6.*

*Para un hombre cuya edad sea  $\geq 36$ , es de \$5.50.*

*Cualquier otro rango de edad o género es un error, y se considera una prima de cero.*

El código para este programa es:

```
public double CalcPrima(double edad, string género)
{
    double prima;
    if (género == "femenino")
    {
        if ((edad >= 18) && (edad <= 30))
        {
            prima = 5.0;
        }
        else
        {
            if (edad >= 31)
            {
                prima = 3.50;
            }
            else
            {
                prima = 0;
            }
        }
    }
    else
    {
        if (género == "masculino")
        {
            if ((edad >= 18) && (edad <= 35))
            {
                prima = 6.0;
            }
            else
            {
                if (edad >= 36)
                {
                    prima = 5.5;
                }
            }
        }
    }
}
```

```
        else
        {
            prima = 0;
        }
    }
    else
    {
        prima = 0;
    }
}
return prima;
}
```

### SOLUCIONES A LAS PRÁCTICAS DE AUTOEVALUACIÓN

**22.1** La especificación no indica qué debe ocurrir cuando surge una excepción. Hay varias posibilidades. La primera situación es si el usuario introduce datos que no sean un entero válido; por ejemplo, que escriba una letra en vez de un dígito. La siguiente situación es si el usuario introduce un número mayor que 10,000. La eventualidad final que podría surgir es si la suma de los números excede el tamaño del número que la computadora puede manejar. Si los enteros se representan como tipos `int` en el programa, este límite es enorme, pero podría darse el caso.

**22.2** Un número de fila puede estar en tres particiones:

1. dentro del rango de 1 a 8
2. entre los menores de 1
3. entre los mayores de 8

Si elegimos un valor representativo en cada partición (digamos, 3, -3 y 11, respectivamente) y un conjunto similar de valores para los números de columna (por ejemplo, 5, -2 y 34), los datos de prueba serán:

Número de prueba	Fila	Columna	Resultado
1	3	5	Correcto
2	-3	5	Inválido
3	11	5	Inválido
4	3	-2	Inválido
5	-3	-2	Inválido
6	11	-2	Inválido
7	3	34	Inválido
8	-3	34	Inválido
9	11	34	Inválido

Ahora debemos considerar que los datos cerca del límite de las particiones son importantes y, por lo tanto, los agregamos a los datos de prueba para cada partición, de manera que tenemos lo siguiente:

1. dentro del rango de 1 a 8 (digamos, 3)
2. menor de 1 (digamos, -3)
3. mayor de 8 (digamos, 11)
4. valor de límite 1
5. valor de límite 8
6. valor de límite 0
7. valor de límite 9

Esto nos da muchas más combinaciones que podemos usar como datos de prueba.

**22.3** Hay cuatro rutas a seguir en el programa, las cuales podemos recorrer con los siguientes datos de prueba:

Número de prueba			Resultado	
1	3	2	1	3
2	3	2	5	5
3	2	3	1	3
4	2	3	5	5

**22.4** Hay tres rutas a seguir en el extracto del programa, incluyendo aquella en la que ninguna de las condiciones de las instrucciones `if` es verdadera. Pero cada uno de los mensajes de error puede desencadenarse con base en dos condiciones. Por lo tanto, los datos de prueba apropiados son:

Número de prueba	Fila	Columna	Resultado
1	5	6	Correcto
2	0	4	Inválido
3	9	4	Inválido
4	5	9	Inválido
5	5	0	Inválido



# Interfaces

**En este capítulo conoceremos:**

- cómo utilizar interfaces para describir la estructura de un programa;
- cómo utilizar interfaces para asegurar la interoperabilidad de las clases dentro de un programa;
- una comparación entre las interfaces y las clases abstractas.

## ● Introducción

---

C# posee una notación para describir la apariencia externa de una clase, a lo cual llamamos *interfaz*. Una interfaz es casi igual que la descripción de una clase, sólo que se omiten los cuerpos de sus propiedades y métodos. No debemos confundir el uso que hacemos aquí de la palabra interfaz con la misma palabra que se utiliza en el término interfaz gráfica de usuario (GUI). Las interfaces tienen dos usos:

- en el diseño;
- para promover la interoperabilidad.

## ● Interfaces para el diseño

---

Con frecuencia se hace hincapié en la importancia que tiene el diseño durante la planeación inicial de un programa. Para ello hay que diseñar todas las clases del mismo. Una forma de documentar el diseño consiste en escribir en términos coloquiales una especificación de los nombres de las clases, sus propiedades y métodos. Pero también es posible escribir esta descripción en sintaxis de C#. Por ejemplo, la interfaz para la clase `Globo` es:



```
public interface GloboInterfaz
{
    void CambiarTamaño(int nuevoDiámetro);
    int CoordX (get; set;);
    void Mostrar (Graphics áreaDibujo);
}
```

Cabe mencionar que omitimos la palabra `class` en la descripción de las interfaces. Además, los métodos y propiedades no se declaran como `public` (ni de ninguna otra manera).

En una interfaz sólo describimos las propiedades, los nombres y parámetros de los métodos, y omitimos sus cuerpos. Observe la gramática que se utiliza para especificar que debemos proporcionar las operaciones `set` y `get` para la propiedad `CoordX`. Una interfaz describe una clase, pero no dice cómo debemos implementar las propiedades, métodos y elementos de datos. Por ende, únicamente describe los servicios que proporciona; así, la interfaz —representa la apariencia externa de una clase desde el punto de vista de sus usuarios (o de un objeto que se crea como instancia de esa clase). En consecuencia, también nos dice qué debe proporcionar la persona que implemente esa clase.

Podemos compilar una interfaz junto con cualquier otra clase, pero en definitiva no podemos ejecutarla. Sin embargo, alguien que planea usar una clase puede compilar el programa junto con la interfaz, y comprobar así que se utilice de manera correcta. Cualquiera que haya escrito un programa en C# sabe que el compilador examina el programa con gran cautela para detectar errores que, de pasar desapercibidos, podrían ocasionar problemas al momento de ejecutarlo. Por lo tanto, cualquier comprobación que deba realizarse en tiempo de compilación bien vale la pena.

El programador puede especificar en el encabezado de la clase que va a implementar cierta interfaz particular. En el ejemplo anterior escribimos una interfaz para la clase `Globo`; ahora escribiremos la clase `Globo` en sí:

```
public class Globo : GloboInterfaz
{
    private int diámetro, x, y;
    public void CambiarTamaño(int nuevoDiámetro)
    {
        diámetro = nuevoDiámetro;
    }
    public int CoordX
    {
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }
}
```

```
public void Mostrar(Graphics áreaDibujo)
{
    Pen lápiz = new Pen(Color.Black);
    áreaDibujo.DrawEllipse(lápiz, x, y, diámetro, diámetro);
}
}
```

Para describir la clase como un todo decimos que extiende la interfaz `GloboInterfaz`. El compilador debe entonces comprobar que esta clase se implemente de manera que cumpla con la declaración de la interfaz; es decir, que proporcione las propiedades y métodos `CambiarTamaño`, `CoordX` y `Mostrar`, junto con sus parámetros apropiados. La regla establece que si implementamos una interfaz, tenemos que implementar también todas las propiedades y métodos descritos en ella. De lo contrario se producirán errores de compilación.

Asimismo podemos usar las interfaces para describir una estructura de herencia. Por ejemplo, suponga que deseamos describir una interfaz para un tipo `GloboColoreado`, que es una subclase de la interfaz `Globo` antes descrita. Podemos escribir:

```
public interface GloboColoreadoInterfaz
    : GloboInterfaz
{
    void SetColor(Color c);
}
```

Esta interfaz hereda las características de la interfaz `GloboInterfaz`, e incluye un método adicional para establecer el color de un objeto. Podríamos describir de manera similar toda una estructura de clases tipo árbol como interfaces, aludiendo solamente a su apariencia externa y a sus relaciones subclase-superclase.

En resumen, es posible utilizar interfaces para describir:

- las clases que conforman un programa;
- la estructura de herencia presente en un programa, es decir, las relaciones del tipo “es un”.

Lo que las interfaces *no pueden* describir es lo siguiente:

- las implementaciones de los métodos y propiedades (esto es, para lo que las interfaces fueron creadas);
- qué clases utilizan otras clases, es decir, las relaciones de tipo “tiene un” (para esto se requiere alguna otra notación).

Para usar las interfaces en el diseño de un programa debemos escribirlas antes de empezar a codificar la implementación de las clases.

Las interfaces son especialmente útiles en programas de mediano y gran tamaño, en los que se utilizan varias clases. En programas extensos que requieren equipos de programadores, su uso es casi imprescindible para facilitar la comunicación entre los miembros del equipo. Las interfaces complementan los diagramas de clases como documentación del diseño del programa.

## ● Interfaces e interoperabilidad

Los dispositivos domésticos, como tostadores y hornos eléctricos, tienen un cable de alimentación con un enchufe en el extremo. El diseño del enchufe es estándar (en cada país), y asegura que el dispositivo pueda utilizarse en cualquier parte (de ese país). En otras palabras, la adopción de una interfaz común asegura la interoperabilidad. En C# podemos utilizar las interfaces de manera similar, para garantizar que los objetos exhiban una interfaz común. Al trabajar con un objeto de este tipo en cualquier parte del programa, podemos estar seguros de que soporta todas las propiedades y métodos especificados por la descripción de la interfaz.

Como ejemplo declararemos una interfaz llamada `Visualizable`. Cualquier clase que cumpla con esta interfaz debe incluir un método llamado `Mostrar`, cuyo propósito es desplegar el objeto en pantalla. La declaración de la interfaz es:

```
public interface Visualizable
{
    void Mostrar(Graphics áreaDibujo);
}
```

El siguiente paso es escribir una nueva clase llamada `Cuadrado`, la cual representa objetos gráficos cuadrados. En el encabezado de la clase indicamos que extiende la interfaz `Visualizable`. Dentro del cuerpo de la clase incluimos el método `Mostrar`:

```
public class Cuadrado : Visualizable
{
    private int x, y, tamaño;

    public void Mostrar(Graphics áreaDibujo)
    {
        Pen lápiz = new Pen(Color.Black);
        áreaDibujo.DrawRectangle(lápiz, x, y, tamaño, tamaño);
    }

    // otros métodos y propiedades de la clase Cuadrado
}
```

Como se indica en el encabezado, esta clase (y cualquier objeto creado a partir de ella) implementa a la interfaz `Visualizable`. Esto significa que podemos trabajar con cualquier objeto de esta clase en un programa, y cuando necesitemos mostrarlo en pantalla podremos usar su método `Mostrar` sin problema alguno.

Por último, cabe mencionar que así como una televisión tiene interfaces para una fuente de energía y para el origen de la señal, C# también permite especificar que una clase implementa varias interfaces. De esta manera, aun cuando una clase sólo puede heredar de otra clase, puede implementar cualquier cantidad de interfaces.

## Fundamentos de programación

Las interfaces y las clases abstractas son similares. En el capítulo 11, que trata sobre la herencia, describimos las clases abstractas. El propósito de una clase abstracta es describir las características comunes de un grupo de clases en forma de superclase, para lo cual se utiliza la palabra clave **abstract**. Las diferencias entre las clases abstractas y las interfaces son:

- Las clases abstractas suelen proporcionar la implementación de algunos de los métodos y propiedades. Por el contrario, una interfaz nunca describe la implementación.
- Una clase puede implementar más de una interfaz, pero sólo es capaz de heredar de una clase abstracta.
- Una interfaz se utiliza en tiempo de compilación para realizar comprobaciones. Por el contrario, una clase abstracta implica la herencia, para lo cual hay que enlazar el método o propiedad apropiados en tiempo de ejecución.
- Una clase abstracta requiere que las clases que la extiendan implementen sus métodos y propiedades **abstract**. De hecho se espera que la clase abstracta sea heredada en algún momento. Pero una interfaz simplemente especifica el esqueleto de una clase, sin implicar que se va a utilizar herencia.

## Errores comunes de programación

- Una clase sólo puede heredar de otra clase, incluyendo una clase abstracta.
- Una clase puede implementar cualquier cantidad de interfaces.

## Nuevos elementos del lenguaje

- **interface** – la descripción de la interfaz externa para una clase que tal vez no se vaya a escribir todavía.
- **:** – se utiliza en el encabezado de una clase para especificar que la clase, propiedad o método implementa una interfaz específica.

## Resumen

- Las interfaces se utilizan para describir los servicios que proporciona una clase.
- Las interfaces son útiles para describir la estructura de un programa. El compilador de C# puede comprobar esta descripción.
- Las interfaces pueden utilizarse para asegurar que una clase se desarrolla conforme a una interfaz específica; en otras palabras, que hay soporte para la interoperabilidad.

**EJERCICIOS****Las interfaces como descripciones de diseño**

- 23.1** Escriba una interfaz para describir ciertas propiedades y métodos de la clase `TextBox` del cuadro de herramientas.
- 23.2** Escriba una interfaz para describir una clase que represente cuentas bancarias. La clase deberá llamarse `Cuenta`. Debe tener los métodos `Depositar` y `Retirar`, junto con una propiedad llamada `SaldoActual`. Decida los parámetros apropiados para los métodos.
- 23.3** Escriba interfaces para describir la estructura de un programa que consiste en varias clases, como el programa `Invasor del ciberespacio`, descrito en el capítulo 20, que habla sobre diseño.

**Interfaces para interoperabilidad**

- 23.4** Escriba una clase llamada `Círculo`, que describa objetos elípticos, e implemente la interfaz `Visualizable` antes descrita.

# Polimorfismo

En este capítulo conoceremos:

- cómo usar el polimorfismo;
- cuándo utilizar el polimorfismo.

## ● Introducción

---

En este capítulo le presentaremos el concepto de polimorfismo mediante un ejemplo sencillo. Suponga que tenemos dos clases, llamadas **Cuadrado** y **Círculo**, respectivamente. Podemos crear una instancia de **Cuadrado** y una instancia de **Círculo** de la forma usual:

```
Cuadrado cuadrado = new Cuadrado();  
Círculo círculo = new Círculo();
```

Suponga que cada clase cuenta con un método llamado **Mostrar**. Entonces podemos mostrar los objetos con las siguientes instrucciones:

```
cuadrado.Mostrar(papel);  
círculo.Mostrar(papel);
```

Aunque estas invocaciones son muy similares, en cada caso invocamos la versión apropiada de **Mostrar**. En realidad, hay dos métodos con el mismo nombre (**Mostrar**), pero son distintos. El sistema de C# se asegura de que siempre se seleccione el método correcto. De esta forma, cuando invocamos el método **Mostrar** para el objeto **cuadrado**, se hace una invocación al método definido dentro de la clase **cuadrado**, y cuando lo hacemos para el objeto **círculo** se hace una invocación al método definido dentro de la clase **círculo**. Ésta es la esencia del polimorfismo.

## ● El polimorfismo en acción

En este capítulo utilizaremos como ejemplo un programa que despliega formas gráficas (cuadrados y círculos) dentro de un cuadro de imagen (Figura 24.1). Cada tipo de forma se describe mediante una clase. En este caso hay una clase llamada **Cuadrado**, y otra llamada **Círculo**.

El programa también utiliza una clase abstracta llamada **Forma** (en el capítulo 11, en donde hablamos sobre la herencia, se explicaron las clases abstractas). Una ventaja de emplear una clase abstracta de este tipo, es que incorpora todas las características compartidas de las formas gráficas. Así, cada subclase hereda estas características compartidas. En este programa no hay muchas características en realidad —sólo la posición de las formas dentro del cuadro de imagen—, pero en general la herencia puede ayudar a reducir el tamaño de las clases y a hacerlas más claras. La clase **Forma** se declara como **abstract** debido a que nunca necesitaremos crear una instancia a partir de ella, sólo de sus subclases.

Además, como veremos en breve, usar una clase abstracta nos ayuda a explotar el polimorfismo.

He aquí la clase abstracta **Forma**.

```
public abstract class Forma
{
    protected int x, y;
    protected int tamaño = 20;
    protected Pen miLápiz = new Pen(Color.Black);
    public abstract void Mostrar (Graphics áreaDibujo);
}
```

Cada forma se describe mediante su propia clase, una subclase de la clase **Forma** (Figura 24.2). Además, cada clase tiene su propio método **Mostrar** para dibujarse a sí misma.

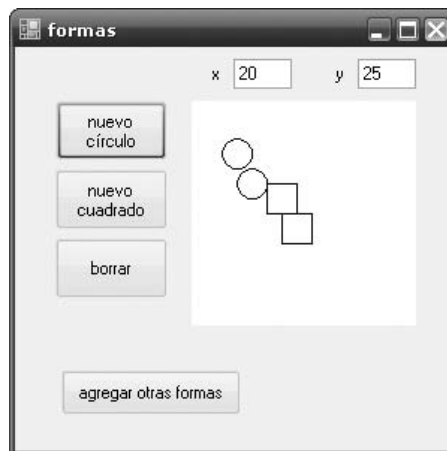


Figura 24.1 Visualización de las formas mediante el uso del polimorfismo.

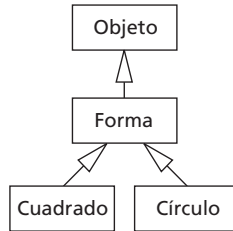


Figura 24.2 Diagrama de clase de las clases de formas.

```
public class Círculo : Forma
{
    public Círculo(int xInic, int yInic)
        : base()
    {
        x = xInic;
        y = yInic;
    }
    public override void Mostrar(Graphics áreaDibujo)
    {
        áreaDibujo.DrawEllipse(miLápiz, x, y, tamaño, tamaño);
    }
}

public class Cuadrado : Forma
{
    public Cuadrado(int xInic, int yInic)
        : base()
    {
        x = xInic;
        y = yInic;
    }
    public override void Mostrar(Graphics áreaDibujo)
    {
        áreaDibujo.DrawRectangle(miLápiz, x, y, tamaño, tamaño);
    }
}
```

Ahora debemos crear una estructura de datos para guardar varias formas. En el capítulo 13 vimos que una lista es una estructura de datos conveniente, que se expande o contrae para dar cabida a los datos. Llamaremos a nuestra lista **grupo**, y la declaramos así en el encabezado del programa:

```
private List <Forma> grupo = new List <Forma>();
```



Esta lista, que en un principio está vacía, es capaz de guardar objetos de la clase `Forma`. Pero afortunadamente también puede almacenar objetos de cualquier subclase de `Forma`, como un objeto `Cuadrado` o `Círculo`. Esto nos demuestra la utilidad de crear la superclase `Forma` y utilizar la herencia.

Para continuar proporcionaremos al usuario la herramienta para crear y agregar algunos objetos a la lista. He aquí el código para que un manejador de eventos responda a un clic en el botón `agregarNuevoCuadrado`:

```
private void agregarNuevoCuadrado_Click(object sender, EventArgs e)
{
    int x, y;
    get(out x, out y);
    Cuadrado cuadrado = new Cuadrado(x, y);
    grupo.Add(cuadrado);
    mostrarTodo();
}
```

Este método obtiene las coordenadas `x` y `y` del usuario (quien las introduce mediante cuadros de texto). Después crea un objeto `cuadrado` y agrega el objeto a la lista.

El código para el botón `agregarNuevoCírculo` es similar:

```
private void agregarNuevoCírculo_Click(object sender, EventArgs e)
{
    int x, y;
    get(out x, out y);
    Círculo círculo = new Círculo(x, y);
    grupo.Add(círculo);
    mostrarTodo();
}
```

A continuación escribiremos el código para mostrar todos los objetos que forman la lista `grupo` dentro de un cuadro de imagen. Los resultados del programa se muestran en la Figura 24.1. El código es:

```
private void mostrarTodo()
{
    Graphics papel = pictureBox1.CreateGraphics();
    foreach (Forma forma in grupo)
    {
        forma.Mostrar(papel);
    }
}
```

Este método utiliza un ciclo `foreach` para acceder a cada uno de los objetos localizados en la lista `group`. Después se usa el polimorfismo: el método `Mostrar` es invocado en varias ocasiones (dentro del ciclo `foreach`) con distintos resultados, de acuerdo con el objeto que se esté utilizando. En la Figura 24.1 podemos ver que las invocaciones de `Mostrar`:

```
forma.Mostrar(papel);
```

producen distintos resultados. Esto no es necesariamente lo que alguien podría esperar, pero es completamente correcto.

En algunas ocasiones, cuando invocamos el método **Mostrar** la variable **forma** contiene un objeto **círculo**, y por lo tanto se invoca la versión de **Mostrar** de la clase **círculo**. Otras veces podría tratarse de un objeto **cuadrado**, con la invocación correspondiente.

Se muestran distintos resultados debido a que el sistema de C# selecciona de manera automática la versión de **Mostrar** asociada con la clase del objeto (y no con la clase de la variable que hace referencia al objeto). La clase de un objeto se determina al momento de crearlo mediante el uso de **new**, y sigue siendo la misma sin importar lo que le ocurra al objeto. Sin importar lo que usted haga con un objeto en un programa, siempre retendrá las características que tenía al momento en que fue creado. Un objeto se puede asignar a una variable de otra clase y pasarse por todo el programa como un argumento, pero nunca perderá su verdadera identidad. El eslogan apropiado podría ser: “un cuadrado siempre será un cuadrado”. Haciendo una analogía con la familia, usted retiene su identidad y su relación con sus antepasados aunque se case, cambie su nombre, cambie de país o haga cualquier otra cosa. Sin duda esto tiene sentido.

Al invocar un método (o usar una propiedad), el polimorfismo se asegura de seleccionar la versión apropiada de ese método (o propiedad). Cuando programamos en C# pocas veces somos conscientes de que el sistema está seleccionando el método correcto para la invocación. Es un proceso automático e invisible.

### Adición de objetos

Como vimos en este pequeño ejemplo, con frecuencia el polimorfismo ayuda a reducir el tamaño de un segmento del programa y lo hace más claro, gracias a que elimina una serie de instrucciones **if**. Pero este logro es mucho más importante de lo que parece; implica que una instrucción como:

```
forma.Mostrar(papel);
```

desconoce por completo la posible variedad de objetos que pueden utilizarse como valor para **forma**. Por lo tanto, se extiende el ocultamiento de la información (que ya está presente en gran medida en cualquier programa orientado a objetos).

Para comprobar esto podemos evaluar cuánto código tendríamos que cambiar en este programa para agregar algún nuevo tipo de forma (una subclase adicional de **Forma**), digamos, un **elipse**. Lo que necesitamos, es:

1. Escribir una nueva clase llamada **Elipse** (una subclase de **Forma**).
2. Agregar un nuevo botón a la interfaz de usuario (para permitir la adición de una forma tipo **elipse**).
3. Escribir el código para crear un nuevo objeto **elipse**, y añadirlo a la lista.

Y eso es todo. No es preciso que modifiquemos el código existente. Por lo tanto, el polimorfismo mejora la modularidad, la capacidad de reutilización, y facilita el mantenimiento.

### Conversión de tipos

Esta sección trata sobre cómo evitar el polimorfismo mediante el uso de código extenso y burdo. Por lo tanto, tal vez desee omitir su lectura y pasar a la siguiente.

Como hemos visto, el polimorfismo nos permite escribir una sola y poderosa instrucción como:

```
forma.Mostrar(papel);
```

Esta instrucción se adapta a cualquier objeto que pertenezca a la clase `Forma`. Una alternativa (menos efectiva) sería utilizar una serie de instrucciones `if`, como se muestra a continuación:

```
if (forma is Círculo)
{
    Círculo círculo = (Círculo) forma;
    círculo.Mostrar(papel);
}
if (forma is Cuadrado)
{
    Cuadrado cuadrado = (Cuadrado) forma;
    forma.Mostrar(papel);
}
```

Sin duda este código es torpe y bastante largo. Utiliza varias instrucciones `if` para diferenciar los tipos de los objetos. Para lograrlo emplea la característica `is` de C#, la cual evalúa si un objeto pertenece a una clase específica.

Este fragmento de programa también aprovecha la conversión de tipos. Recuerde que un objeto siempre mantiene su identidad, es decir, la clase a partir de la cual fue creado. Por ejemplo, la instrucción:

```
Círculo círculo = new Círculo(20, 20);
```

crea un objeto `circulo` de la clase `Círculo`. Pero podemos colocar este objeto en una lista diseñada para guardar objetos de la superclase `Forma`; esto se logra así:

```
grupo.Add(circulo);
```

Lo que ocurre aquí es que se coloca en la lista una referencia (o apuntador) a `circulo`. En el capítulo 5 hablamos de las referencias, cuando analizamos el paso de argumentos por referencia. Pero ahora queremos usar el método `Mostrar` en un objeto `Círculo`. Por lo tanto, necesitamos hacer lo siguiente:

```
Círculo círculo = (Círculo) forma;
```

A esto se le llama conversión de tipos, y significa convertir una referencia en una referencia a una clase distinta. En el código anterior, `forma` es un objeto de la clase `Forma`. La expresión `(Círculo) forma` convierte esta referencia en una referencia a un objeto `Círculo`. Es un concepto bastante denso, pero necesitamos asegurar que se mantenga un estricto manejo de tipos para que el compilador esté feliz.

Si este programa tuviera una gran cantidad de formas, habría un número igualmente elevado de instrucciones `if` y operaciones de conversión de tipos. El polimorfismo evita todo esto, lo que nos deja ver cuán poderoso y conciso es.

## Enlace tardío

El ejemplo práctico que vimos antes utiliza una diversidad de objetos (formas) cuyas características comunes se incorporan a una superclase. Ahora sabemos que la herencia nos ayuda a describir con eficiencia la similitud de grupos de objetos. El otro lado de la moneda es *utilizar* los objetos, y aquí

es en donde el polimorfismo nos ayuda a utilizar objetos de manera uniforme y concisa. La diversidad se maneja no mediante una proliferación de instrucciones `if`, sino a través de una sola invocación a un método:

```
forma.Mostrar(papel);
```

De manera que se hace uso del polimorfismo para seleccionar el método apropiado. Cuando esto ocurre, se selecciona la versión del método `Mostrar` que coincide con el objeto en sí. Esta decisión sólo puede tomarse cuando el programa está en ejecución, justo antes de invocar el método, acción conocida como *enlace tardío*, *enlace dinámico* o *vinculación retrasada*. Se trata de una característica esencial de un lenguaje que soporta el polimorfismo.

### Cuándo debemos usar el polimorfismo

En general, la metodología para explotar el polimorfismo dentro de un programa específico es la siguiente:

1. Use los mismos nombres para los métodos y propiedades similares.
2. Identifique las similitudes (métodos, propiedades y variables comunes) entre los objetos o clases que conforman el programa.
3. Diseñe una superclase que englobe las características comunes de las clases.
4. Diseñe las subclases que describan las características distintivas de cada una de las clases, al tiempo que hereden las características comunes de la superclase.
5. Identifique cualquier punto del programa en donde se deba aplicar la misma operación a cualquiera de los objetos similares. Tal vez se vea tentado a utilizar instrucciones `if` en esta ubicación; sin embargo, es el mejor lugar para usar el polimorfismo.
6. Asegúrese de que la superclase contenga un método (o propiedad) abstracto que corresponda con cada método (o propiedad) que se vaya a utilizar mediante el polimorfismo.

## Fundamentos de programación

El polimorfismo representa el tercer elemento más importante de la programación orientada a objetos. El conjunto completo de estos elementos es:

1. Encapsulamiento: los objetos pueden hacerse altamente modulares.
2. Herencia: las características deseables en una clase existente se pueden reutilizar en otras clases sin afectar la integridad de la clase original.
3. Polimorfismo: consiste en diseñar código que pueda manipular con facilidad objetos de distintas clases. Las diferencias entre los objetos similares pueden ajustarse con facilidad.

Por lo general los programadores principiantes empiezan utilizando el encapsulamiento, más tarde pasan a la herencia, y después emplean el polimorfismo.

El polimorfismo ayuda a construir programas:

- concisos (más cortos de lo que serían si no se utilizara);
- modulares (las partes no relacionadas se mantienen separadas);
- fáciles de modificar y adaptar (por ejemplo, cuando se introducen nuevos objetos).

## Errores comunes de programación

Si piensa explotar el polimorfismo para agrupar varias clases bajo una sola superclase, asegúrese de que la superclase describa todos los métodos y propiedades que se utilizarán en cualquier instancia de la superclase. Algunas veces para ello se requieren métodos y propiedades abstractos en la superclase, cuyo único propósito es permitir que el programa compile.

## Nuevos elementos del lenguaje

La palabra clave `is` permite que el programa evalúe si un objeto pertenece a una clase específica.

## Resumen

Los principios del polimorfismo son:

1. Un objeto siempre retiene la identidad de la clase a partir de la cual fue creado (nunca podrá convertirse en un objeto de otra clase).
2. Cuando se utiliza un método o propiedad en un objeto, se selecciona de manera automática el método o propiedad correctos.

El polimorfismo ayuda al ocultamiento de información y reutilizar código, ya que nos ayuda a que las piezas de código tengan un amplio rango de aplicación.

## EJERCICIOS

- 24.1** Una clase abstracta llamada `Animal` tiene un método constructor, una propiedad llamada `Peso` y un método denominado `Dice`. La propiedad `Peso` representa el peso del animal, un valor `int`. El método `Dice` devuelve una cadena, el ruido que hace el animal. La clase `Animal` tiene las subclases `Vaca`, `Serpiente` y `Cerdo`. Estas subclases realizan distintas implementaciones de `Dice`, las cuales devuelven los valores "mu", "sss" y "oinc", respectivamente. Escriba la clase `Animal` y las tres subclases. Cree los objetos `mascotaVaca`, `mascotaSerpiente` y `mascotaCerdo` a partir de las clases correspondientes, y utilice sus propiedades y métodos. Despliegue la información en un cuadro de texto.
- 24.2** Agregue una nueva forma (una línea recta) a la colección de formas disponibles en el programa `Formas`. Use el método de biblioteca `DrawLine` para dibujar un objeto línea. Añada código para crear un objeto línea, agregarlo a la lista de objetos (en la lista), y desplegarlo en pantalla junto con las otras formas.
- 24.3** Mejore el programa `Formas` para convertirlo en un paquete de dibujo completo, que permita seleccionar formas de un botón y colocarlas en el punto deseado dentro de un cuadro de

imagen. El usuario debe especificar la posición con un clic del ratón. También debe permitirle que borre las formas.

**24.4** Un banco ofrece a sus clientes dos tipos de cuentas: la regular y la dorada. Ambos tipos de cuentas proveen varias características compartidas, pero también ofrecen otras distintivas. Las características comunes son:

- registrar el nombre y la dirección;
- abrir una cuenta con un saldo inicial;
- mantener y mostrar un registro del saldo actual;
- métodos para depositar y retirar una cantidad de dinero.

La cuenta regular no puede sobregirarse. El titular de una cuenta dorada puede sobregirarse de manera indefinida. En una cuenta regular el interés se calcula a razón de 5% anual. La cuenta dorada tiene una tasa de interés de 6% anual, menos un cargo fijo de \$100 al año.

Escriba una clase que describa las características comunes, y escriba clases para describir las cuentas regular y dorada.

Construya un programa para utilizar estas clases. El programa debe crear dos cuentas bancarias: una regular y otra dorada. Cada cuenta deberá crearse con el nombre de una persona y cierta cantidad de dinero. Despliegue en pantalla el nombre, el saldo y el interés de cada cuenta.

## Apéndice A

# Componentes de la biblioteca seleccionados

En este apéndice se resume información sobre:

- los componentes utilizados en todo el libro;
- componentes adicionales del cuadro de herramientas de C#. Muchos de ellos se ilustran en la Figura A.1.

El análisis de cada componente está acompañado por un resumen de sus propiedades, métodos y eventos importantes. Para obtener más detalles sobre los componentes busque en el sistema de ayuda de C#, o utilice la biblioteca de documentación de Microsoft en:

`msdn.microsoft.com/library/`

Los componentes que comentaremos aquí son:

Button	ListBox	RichTextBox
CheckBox	MenuStrip	StatusStrip
ComboBox	MonthCalendar	TabControl
DateTimePicker	NumericUpDown	TextBox
Graphics	Panel	Timer
GroupBox	PictureBox	ToolStrip
Label	ProgressBar	ToolTip
LinkLabel	RadioButton	TrackBar
List	Random	

Presentaremos la lista de componentes en dos secciones: los controles de GUI del cuadro de herramientas, y el resto, que son `Random`, `List` y `Graphics`.

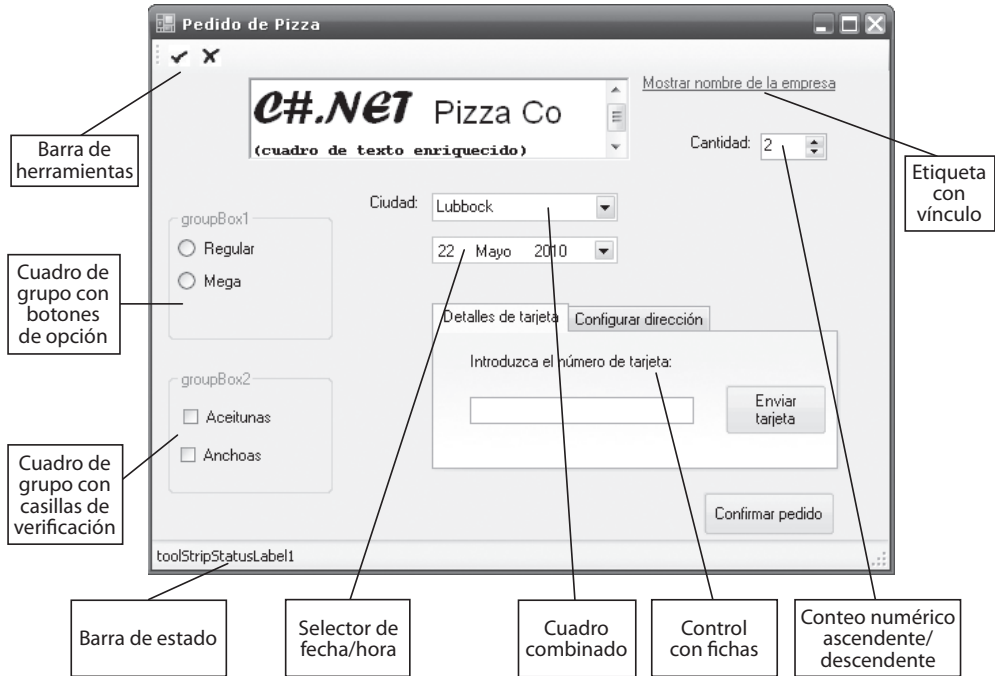


Figura A.1 El programa Pedido de pizza.

## ● Los controles del cuadro de herramientas

En esta sección veremos un resumen de los principales controles de GUI. La Figura A.1 muestra un formulario que contiene muchos de esos controles y permite al usuario seleccionar opciones relacionadas con un pedido de pizza. Al hacer clic en **Confirmar pedido** se muestra la elección del usuario en cuadros de mensaje.

He aquí el código del evento para el botón **Confirmar pedido**, al cual nos referiremos más adelante:

```
private void botónConfirmarButton_Click(object sender, EventArgs e)
{
    if (regularRadioButton.Checked)
    {
        MessageBox.Show("Usted eligió una pizza regular");
    }
    else
    {
        if (megaRadioButton.Checked)
        {
            MessageBox.Show("Usted eligió una mega pizza");
        }
    }
}
```



```

    }
}

if (anchoasCheckBox.Checked)
{
    MessageBox.Show("Usted quiere anchoas en su pizza");
}
if (aceitunasCheckBox.Checked)
{
    MessageBox.Show("Usted quiere aceitunas en su pizza");
}
MessageBox.Show("Fecha/Hora: " +
    Convert.ToString(dateTimePicker1.Value));
MessageBox.Show("Cantidad: " +
    Convert.ToString(cantidadNumericUpDown.Value));
MessageBox.Show("Número de ciudad: " +
    Convert.ToString(ciudadComboBox.SelectedIndex));
MessageBox.Show("Nombre de la ciudad: " +
    ciudadComboBox.Items[ciudadComboBox.SelectedIndex]);
toolStripStatusLabel1.Text = "Confirmación recibida";
}

```

## ● Características que proporcionan todos los controles de GUI

Todos los componentes de GUI comparten ciertas propiedades, métodos y eventos comunes, tal como se ilustra a continuación.

### Propiedades

<b>Top</b>	Contiene la coordenada $y$ de la parte superior del componente.
<b>Left</b>	Contiene la coordenada $x$ del lado izquierdo del componente.
<b>Width</b>	Es el ancho del componente.
<b>Height</b>	Es la altura del componente.
<b>Visible</b>	Determina si el componente es visible o no cuando el programa está en ejecución.
<b>Enabled</b>	Especifica si el componente puede utilizarse ( <b>true</b> ) o si está deshabilitado ( <b>false</b> ).
<b>MousePosition</b>	Contiene las coordenadas actuales del cursor del ratón relativo a un control. Ejemplo: <pre>int x = pictureBox1.MousePosition.X; int y = pictureBox1.MousePosition.Y;</pre>

## Métodos

<b>Focus</b>	Para los controles que permiten introducir datos ( <b>Button</b> , <b>TextBox</b> , etc.). <b>Focus</b> hace que se seleccione el control especificado, de manera que el usuario no tenga que mover el cursor del ratón para seleccionarlo. Devuelve un valor <b>bool</b> de <b>true</b> si se aceptó la invocación. Ejemplo: <pre>bool aceptado = button1.Focus();</pre>
<b>CreateGraphics</b>	Crea un objeto de dibujo <b>Graphics</b> en el control especificado. Ejemplo: <pre>pictureBox1.CreateGraphics();</pre>

## Eventos

<b>MouseMove</b>	Se invoca cuando desplazamos el cursor del ratón sobre un control.
<b>Click</b>	Se invoca cuando el usuario hace clic en un componente. Más adelante describiremos este evento para los controles que lo requieran.
<b>DoubleClick</b>	Se invoca cuando el usuario hace doble clic en un control.

## ● **Button (botón)**

---

### Propiedades

<b>Text</b>	El texto que se muestra en el botón.
-------------	--------------------------------------

### Eventos

<b>Click</b>	Se invoca cuando el usuario hace clic en el botón.
--------------	--

## ● **CheckBox (casilla de verificación)**

---

Vea la Figura A.1 y el código correspondiente a la selección de los ingredientes de la pizza. Las casillas de verificación proveen la manera de introducir opciones pero, a diferencia de los botones de opción (que veremos más adelante) no son mutuamente exclusivas: el valor de la propiedad **Checked** de cada una de ellas no está conectado a las demás. En muchos casos podríamos optar por no realizar acción alguna cuando ocurre el evento **checkedChanged**. En vez de ello, el código asociado a un botón para confirmar puede examinar la propiedad **checked** de cada casilla de verificación, como se hace en nuestro programa Pedido de pizza. Al adoptar esta forma de trabajar permitimos que el usuario revise sus elecciones antes de confirmarlas.

### Propiedades

<b>Checked</b>	El valor <b>bool</b> de la casilla de verificación.
<b>Text</b>	El texto que aparece a un lado de la casilla.

## Eventos

**CheckedChanged** Se invoca cuando el usuario modifica el estado de la casilla de verificación.

### ● ComboBox (cuadro combinado)

Observe la Figura A.1; ahí podrá ver un espacio en donde se selecciona la ciudad (la elección se mostrará más tarde mediante un cuadro de mensaje). Dicho espacio es un cuadro combinado: una mezcla entre el cuadro de texto y el cuadro de lista. Ocupa menos espacio en pantalla debido a que las selecciones no están permanentemente a la vista.

Una de sus propiedades principales es **DropDownStyle**, que por lo general se establece en tiempo de diseño. Esta propiedad puede adoptar tres estilos:

- Simple: el cuadro no es “desplegable” (en este libro no se usa este estilo).
- Drop-down: el cuadro puede editarse y la lista es desplegable (es el que usamos en el programa Pedido de pizza).
- Drop-down list: la lista es desplegable, pero no se permite editarla.

Los cuadros combinados suelen usarse junto con controles como casillas de verificación para recolectar un conjunto de opciones. Por lo general utilizamos un botón “confirmar” para reunir todas las selecciones que realiza el usuario.

Todos los tipos de cuadro combinado permiten establecer su propiedad **Items** (es decir, determinar de cuáles elementos constará el listado) en tiempo de diseño, y también podemos agregar elementos en tiempo de ejecución.

En el caso del estilo “drop-down” (editable), el único elemento visible puede establecerse mediante la propiedad **Text**; para obtener la cadena que el usuario seleccionó o escribió empleamos código como el siguiente:

```
string ciudad;
ciudad = ciudadComboBox.Text;
```

En el estilo “drop-down list” (no editable) la propiedad **Text** no está disponible. En su lugar, podemos usar el control de la misma forma que usamos un cuadro de lista (consulte el capítulo 13). Es posible obtener la selección actual mediante el siguiente código:

```
string ciudad;
ciudad = ciudadComboBox.Items[CiudadCombo.SelectedIndex];
```

El cuadro combinado utiliza muy poco espacio en pantalla, pero la desventaja surge cuando tenemos varios conjuntos de opciones en un solo formulario. El usuario no podrá ver al mismo tiempo todo el espectro de opciones disponibles.

## Propiedades

**Text** El único elemento visible en el cuadro combinado.

**DropDownStyle** Simple, DropDown, DropDownList.

Los demás miembros principales son idénticos a los de un cuadro de lista, cuyas opciones veremos más adelante.

### ● DateTimePicker (selector de fecha/hora)

---

Vea la Figura A.1. El selector de fecha/hora se parece a un cuadro combinado que contiene una fecha. Al hacer clic en él se abre un objeto calendario mensual, el cual nos permite seleccionar un valor de fecha específico. Cuando se establece la propiedad `ShowUpDown` como `true` es posible seleccionar la fecha y hora mediante una pantalla desplazable, como si se tratara de un control numérico ascendente/descendente.

#### Propiedades

<code>Value</code>	El valor seleccionado (un objeto <code>DateTime</code> ).
<code>ShowUpDown</code>	Aparece en estilo calendario o desplazable.

#### Eventos

<code>ValueChanged</code>	Se invoca al modificar la fecha seleccionada.
---------------------------	---

### ● GroupBox (cuadro de grupo)

---

Vea la Figura A.1; ubique el sitio en donde se agrupan los botones de opción y las casillas de verificación. Éste es el cuadro de grupo, un contenedor para otros controles. Podemos establecer su propiedad `Text` para proveer un nombre descriptivo a los controles que incluyamos en él. Al mover un cuadro de grupo, todos los controles que contiene se desplazan también. Además, proporciona una funcionalidad especial para los botones de opción que colocamos en su interior, ya que trabajan de manera independiente a cualquier otro botón de opción que pudiera localizarse en otra parte del formulario.

### ● Label (etiqueta)

---

#### Propiedades

<code>Text</code>	El texto desplegado en la etiqueta.
-------------------	-------------------------------------

## ● LinkLabel (etiqueta con vínculo)

Localice en la Figura A.1 el sitio donde iniciamos la carga del nombre de la empresa. Ahí usamos una etiqueta con vínculo, que provee texto en el que se puede hacer clic en vez de hacerlo en un botón. A diferencia del botón, el color del texto cambia una vez que se activa el vínculo.

La etiqueta con vínculo puede proporcionar una interfaz estilo Web para las aplicaciones Windows.

### Propiedades

**Text** El texto visible del vínculo.

### Eventos

**Click** Se invoca al hacer clic en el vínculo.

## ● ListBox (cuadro de lista)

### Propiedades

**SelectedIndex** El índice del elemento actual seleccionado (resaltado) en la lista (o `-1` si no hay un elemento seleccionado).

**SelectedItem** El valor de cadena del elemento actual seleccionado en el cuadro de lista.

**Items** Un objeto `List` que contiene todos los elementos que conforman la lista.

**Sorted** Si su valor es `true`, se asegura de que los elementos de la lista se ordenen de manera automática y se mantengan ordenados incluso cuando se agreguen nuevos elementos.

### Eventos

**SelectedIndexChanged** Se invoca cuando el usuario hace clic en un elemento del cuadro de lista.

**DoubleClick** Se invoca cuando el usuario hace doble clic en un elemento del cuadro de lista.

### ● **MenuStrip (barra de menús)**

---

Consulte la información pertinente en el capítulo 18, en donde analizamos la creación de menús.

### ● **MonthCalendar (calendario mensual)**

---

Este control proporciona un calendario a partir del que podemos seleccionar fechas. El selector de fecha/hora también puede configurarse para mostrar un calendario mensual desplegable. En este libro no veremos los detalles relacionados con este control.

### ● **NumericUpDown (conteo numérico ascendente/descendente)**

---

Vea la Figura A.1; el sitio en donde podemos elegir el número de pizzas fue creado con un control `NumericUpDown`, que permite al usuario seleccionar un número de dos formas: escribiéndolo o usando las flechas para desplazar los valores en orden ascendente/descendente. Podemos establecer sus propiedades `Maximum`, `Minimum` e `Increment` según se requiera. Utilizamos su propiedad `Value` (un valor `Decimal`) para acceder a la opción actual.

Cualquier carácter no numérico se rechaza.

Esto puede ser más simple que utilizar un cuadro de texto, en especial cuando el usuario prefiere emplear el ratón.

He aquí cómo podemos acceder a la opción actual:

```
MessageBox.Show("Número: " +  
                Convert.ToString(cantidadNumericUpDown.Value));
```

Cabe mencionar que el tipo `Decimal` puede almacenar hasta 29 dígitos de precisión.

### **Propiedades**

<code>Value</code>	El valor seleccionado (un tipo <code>Decimal</code> ).
<code>Maximum</code>	El número más grande que se puede seleccionar ( <code>Decimal</code> ).
<code>Minimum</code>	El número más pequeño que se puede seleccionar ( <code>Decimal</code> ).

### **Eventos**

<code>ValueChanged</code>	Se invoca al modificar el valor actual.
---------------------------	---

### ● **Panel**

---

El panel es semejante al cuadro de grupo, pero no admite el uso de etiquetas de texto. Puede contener varios botones, con la ventaja de que nos permite moverlos como un conjunto.

## ● PictureBox (cuadro de imagen)

### Propiedades

**Image** La imagen que se despliega en el cuadro. Se puede establecer en tiempo de diseño o en tiempo de ejecución mediante el uso de `FromFile`. Por ejemplo:

```
pictureBox1.Image = Graphics.FromFile("nombreArchivo.jpeg");
```

Además, permite la extracción de la imagen desplegada en el cuadro para copiarla a otro cuadro de imagen o dibujarla dentro de un área de gráficos.

### Eventos

**Click** Se invoca cuando el usuario hace clic en la imagen.

Recuerde que los eventos `MouseMove` y `DoubleClick` se aplican a este control.

## ● ProgressBar (barra de progreso)

Podemos utilizar la barra de progreso para brindar una señal visual del progreso de una tarea que se lleve mucho tiempo, como instalar un programa o copiar una gran cantidad de archivos. Es posible establecer el número de divisiones de que constará la barra mediante las propiedades `Maximum` y `Minimum`; la propiedad `Step` controla qué tanto avanza la barra en cada paso. Por ejemplo, podemos establecer una barra de progreso con intervalos que cambian bruscamente si establecemos la propiedad `Maximum` en 10, la propiedad `Minimum` en 0 y la propiedad `Step` en 1.

Para hacer que la barra avance un paso utilizamos el método `PerformStep`, como en el siguiente ejemplo:

```
progressBar1.PerformStep();
```

El programador puede colocar invocaciones a `PerformStep` en los lugares adecuados durante la realización de una tarea extensa, o invocarlo desde un evento de temporizador.

### Propiedades

**Maximum** El valor máximo de la barra de progreso.  
**Minimum** El valor mínimo de la barra de progreso.  
**Step** El espacio que se desplaza la barra al invocar el método `PerformStep`.

### Métodos

**PerformStep** Hace que la barra avance un paso.

## ● **RadioButton (botón de opción)**

---

El botón de opción permite que el usuario seleccione una sola opción entre un conjunto de alternativas. Los controles `RadioButton` se consideran “mutuamente exclusivos”, en cuanto a que sólo un botón de opción de los que conforman un conjunto puede estar “encendido” (activado) en un momento dado. Podemos colocar texto descriptivo al lado del botón.

Si un formulario tiene varios conjuntos de botones de opción, cada uno de ellos deberá colocarse dentro de un cuadro de grupo para evitar la interacción entre conjuntos de botones. Si no hacemos esto sólo podremos “encender” un botón de opción en todo el formulario.

Cuando un usuario modifica el estado de un botón de opción (lo enciende o lo apaga) ocurre un evento `CheckedChanged` y podemos examinar la propiedad `bool Checked`. Sin embargo, en el programa Pedido de pizza (vea la Figura A.1 y su código asociado) para acceder a la selección examinamos cada uno de los botones de opción cuando el usuario hace clic en el botón “confirmar”.

### **Propiedades**

<code>Checked</code>	Un valor <code>bool</code> que proporciona el estado de un botón de opción específico.
<code>Text</code>	El texto a mostrar a un lado del botón de opción.

### **Eventos**

<code>CheckedChanged</code>	Se invoca al modificar un botón de opción.
-----------------------------	--

## ● **RichTextBox (cuadro de texto enriquecido)**

---

Vea la Figura A.1. El cuadro de texto enriquecido que se ubica en la parte superior del programa provee herramientas similares al cuadro de texto normal, sólo que puede desplegar el texto en diversas fuentes y tamaños al mismo tiempo. Para lograr este efecto podríamos crear el texto en un procesador de palabras y guardarlo en formato de texto enriquecido (RTF). Microsoft creó este formato para permitir que varias marcas de procesadores de palabras pudieran leer los documentos. Para cargar un documento RTF en un cuadro de texto debemos usar una instrucción como la siguiente:

```
richTextBox1.LoadFile("archivodemo.rtf");
```

Este control cuenta con muchas más herramientas que el cuadro de texto (cuyas características comentaremos más adelante) y es preferible a la hora de realizar manipulaciones de texto serias. No veremos este control con detalle.

## ● **StatusStrip (barra de estado)**

---

Vea la Figura A.1. La tira de estado se coloca en la parte inferior de un formulario y se utiliza con frecuencia para mostrar mensajes informativos. Para utilizarla debemos agregar uno de los cuatro elementos disponibles, haciendo clic en la lista desplegable que aparece dentro del control. En el ejemplo del programa Pedido de pizza agregamos un elemento `ToolStripStatusLabel` y lo utilizamos de la siguiente forma:

```
toolStripStatusLabel1.Text = "Confirmación recibida ";
```



## ● TabControl (control con fichas)

Vea la Figura A.1. Hemos visto este control muchas veces, ya que el IDE de C# lo utiliza para alternar entre el diseño de un formulario y su código. Imita las hojas de papel que se traslapan entre sí, cada una con una pestaña que sobresale de su parte superior. Cada hoja es un objeto `TabPage`.

He aquí el proceso de creación:

- Seleccione un control `TabControl` del cuadro de herramientas, y colóquelo en el formulario.
- Abra la colección `TabPages` en la lista de propiedades.
- En el cuadro de Editor de la colección `TabPage` seleccione la primera ficha, llamada `tabPage1`, de la lista `Miembros`.
- Establezca el nombre y la propiedad `Text` de esta ficha según lo requiera. El texto debe ser lo más descriptivo posible, ya que será la única parte de la página que podrá verse en todo momento.
- Repita el proceso anterior para configurar la página `tabPage2`. Si necesita agregar más fichas, haga clic en el botón `Agregar`, si requiere eliminar una haga clic en `Quitar`.
- Por último, agregue controles a cada pestaña. Tal vez quiera incorporar el nombre de la página de un control, en el nombre del control mismo; por ejemplo, `pestañaTarjeta`.

### Propiedades

**Text** El texto que debe aparecer en la pestaña visible de una página.

## ● TextBox (cuadro de texto)

### Propiedades

<b>Text</b>	El texto que contiene el cuadro.
<b>Locked</b>	Determina si se puede mover o modificar el tamaño del cuadro de texto en tiempo de diseño.
<b>MultiLine</b>	Determina si el cuadro puede alojar más de una línea de texto.
<b>ReadOnly</b>	Determina si el usuario puede modificar el texto.
<b>ScrollBars</b>	Opciones para desplegar barras de desplazamiento horizontal y/o vertical.
<b>BackColor</b>	Determina el color de fondo del cuadro.
<b>Font</b>	Determina el estilo de la fuente y el tamaño del texto.

### Métodos

<b>AppendText</b>	Adjunta texto al final del texto actual. Parámetro: <code>string newText</code>
<b>Clear</b>	Vacía el cuadro de texto.

## ● Timer (temporizador)

---

### Propiedades

<b>Enabled</b>	Se establece en <code>true</code> para activar el temporizador.
<b>Interval</b>	Determina la frecuencia de eventos <code>Tick</code> en milisegundos (1/1000 segundos).

### Eventos

<b>Tick</b>	Se invoca cuando el temporizador pulsa cada cierto intervalo de milisegundos, definido por <code>Interval</code> .
-------------	--

## ● ToolStrip (barra de herramientas)

---

El control barra de herramientas contiene varios botones con texto o iconos. Al usar iconos los botones pueden ser pequeños y, por ende, se libera espacio en pantalla para otros componentes. En la Figura A.1 creamos iconos de “paloma” y “tache” () y los agregamos a los botones de la barra de herramientas. He aquí el proceso:

- Use un programa de dibujo para crear sus imágenes, y guarde cada una de ellas en un archivo. El tipo JPEG es apropiado, pero también se permiten otros. El tamaño típico de un icono es de 30 por 30 píxeles.
- Coloque un control `ToolStrip` en el formulario; el control se ubicará de manera automática en la parte superior del mismo.
- Haga clic en la lista desplegable del control `ToolStrip` y agregue un elemento `Button` para sumar un botón a la barra de herramientas.
- Haga clic con el botón derecho del ratón sobre el botón que acaba de agregar, y seleccione la opción **Establecer imagen** del menú desplegable.
- A continuación aparecerá el cuadro de diálogo **Seleccionar recurso**. Haga clic en el botón **Importar...** de la sección **Recurso local** y, en el cuadro de diálogo que aparezca, seleccione la imagen de la “paloma” que creó en el programa de dibujo. Haga clic en **Abrir** después de seleccionarla, y haga clic en **Aceptar** para cerrar el cuadro de diálogo **Seleccionar recurso** y regresar al formulario de la aplicación. La imagen aparecerá en el botón.
- Repita el proceso para el botón con el icono de “tache”.

Al hacer clic en uno de los botones se invoca el método del evento `ButtonClick` del mismo. Por ejemplo, el código siguiente muestra un mensaje cada vez que el usuario hace clic en los botones “paloma” o “tache” de la barra de herramientas:

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
    MessageBox.Show("El usuario hizo clic en el botón 'paloma'");
}
private void toolStripButton2_Click(object sender, EventArgs e)
{
    MessageBox.Show("El usuario hizo clic en el botón 'tache'");
}
```

## ● ToolTip (sugerencia contextual)

Las sugerencias contextuales son mensajes que aparecen cuando colocamos el cursor del ratón sobre un control y lo mantenemos ahí por algunos segundos. Al arrastrar un control `ToolTip` hasta un formulario, C# lo coloca en la bandeja de componentes. Ahora cada uno de los controles del formulario puede tener una sugerencia contextual que explique su función; la manera más sencilla de hacerlo es en tiempo de diseño, manipulando las propiedades de cada control.

## ● TrackBar (barra de seguimiento)

La barra de seguimiento proporciona un “control deslizable” con intervalos definidos, el cual podemos arrastrar con el cursor del ratón para realizar tareas como ajustar el volumen de un altavoz. Su utilización se describe en el capítulo 6.

### Propiedades

<code>Maximum</code>	El valor más grande que el usuario puede seleccionar en la barra de seguimiento.
<code>Minimum</code>	El valor más pequeño que el usuario puede seleccionar en la barra de seguimiento.
<code>Orientation</code>	La orientación de la barra de seguimiento; puede ser horizontal o vertical.
<code>Value</code>	El valor actual de la barra de seguimiento.

### Eventos

<code>Scroll</code>	Se invoca al mover la barra de seguimiento.
---------------------	---

## ● Otros componentes

En esta sección haremos un resumen de los componentes que no son del cuadro de herramientas que utilizamos en este libro.

## ● List (lista)

En el capítulo 13 describimos este componente con detalle.

## Constructor

**List** Ejemplo:  
`List<string> colección = new List<string>();`

## Propiedades

**Count** El número de elementos que conforman la lista.

## Métodos

**Add** Agrega un nuevo objeto a la lista. El nuevo elemento se añade al final de la misma.

**RemoveAt** Devuelve y elimina el objeto en el índice especificado de la lista. El resto de los elementos se desplaza hacia arriba para llenar el vacío que se creó.

**Remove** Elimina el objeto de la lista.

**Insert** Inserta un nuevo objeto en la lista, en la posición del índice especificado. El resto de los elementos se desplaza hacia abajo para hacer espacio al nuevo elemento. Parámetros:  
`int índice, object nuevoElemento`

**Clear** Vacía toda la lista.

## ● Graphics

---

Para crear un área de dibujo de gráficos utilizamos el método `CreateGraphics`, que podemos aplicar a cualquier control. Ejemplo:

```
Graphics papel = pictureBox1.CreateGraphics();
```

## Métodos

**Clear** Borra el área de dibujo y la rellena con el color que se proporciona como parámetro.

**DrawLine** Dibuja una línea. Parámetros:  
`Pen miLápiz, x, y, xFinal, yFinal`

**DrawEllipse** Dibuja una elipse. Parámetros:  
`Pen miLápiz, x, y, ancho, altura`

**DrawImage** Dibuja una imagen. Parámetros:  
`Image miImagen, x, y, ancho, altura`

**DrawRectangle** Dibuja un rectángulo. Parámetros:  
`Pen miLápiz, x, y, ancho, altura`

**FillEllipse** Dibuja una elipse rellena. Parámetros:  
`SolidBrush miPincel, x, y, ancho, altura`

**FillRectangle** Dibuja un rectángulo relleno. Parámetros:  
`SolidBrush miPincel, x, y, ancho, altura`

## ● Random

---

### Constructor

Random

Ejemplo:

```
Random aleatorio = new Random();
```

### Métodos

Next

Devuelve un número aleatorio en el rango de **valorMin** a **valorMax** - 1.

Parámetros:

```
int valorMin, int valorMax
```

## Apéndice B

# Palabras reservadas (clave)

El lenguaje C# tiene ciertas palabras reservadas que no podemos usar para dar nombre a nuestras variables, clases, métodos o propiedades.

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

# Bibliografía

Esta bibliografía es para los lectores que deseen continuar aprendiendo sobre los temas vistos en este libro.

*Refactoring*, **Martin Fowler, Addison-Wesley, 1999.**

El capítulo 20, que habla sobre diseño, menciona la refactorización como una técnica para reordenar los métodos y propiedades en un diseño. Éste es el libro precursor sobre factorización, y lo mejor es que resulta fácil de leer. Con sólo darle una hojeada aprenderá técnicas y conocerá perspectivas útiles.

*Extreme Programming Explained*, **Kent Beck, Addison-Wesley, 2000.**

Este libro complementa nuestros capítulos sobre diseño y prueba. Presenta muchas buenas ideas sobre cómo programar sin estrés.

*UML Distilled*, **Martin Fowler con Kendall Scott, Addison-Wesley, 2000.**

UML es la notación dominante para describir programas. En este libro utilizamos esta notación en la medida apropiada. Uno de los libros más simples de todos los que se han escrito sobre UML.

*About Face*, **Alan Cooper, IDG Books, 1995.**

Alan Cooper es reconocido como la persona que creó el primer IDE de Visual Basic, en donde se podían arrastrar componentes a los formularios de diseño. Este libro contiene sus ideas individuales sobre el diseño de interfaces de usuario. Tal vez no sea una obra que usted lea de principio a fin, pero sus críticas sobre algunas interfaces de usuario y sus sugerencias para mejorar los sistemas harán de su lectura una experiencia reveladora. A diferencia de muchos libros de texto sobre interfaces de usuario, se enfoca en los sistemas Microsoft Windows.

*Goto: Superheroes of Software Programming from Fortran to the Internet Age and Beyond*, **Steve Lohr, Profile Books, 2002.**

Este libro traza la historia de los lenguajes de programación, enfocándose en las contribuciones de los pioneros del software. Cubre temas como Fortran, C, C++, Java, Unix y Visual Basic. Capítulo a capítulo analiza la experiencia de importantes personajes del ámbito, incluye entrevistas con ellos, además de comentarios sobre sus motivaciones y la tecnología que utilizaron.

# Índice

!, 125, 150  
!=, 118, 145  
", 47  
\$, 214  
%, 44, 46  
&&, 125, 150  
( ), 29  
(double), operador de conversión, 53  
(int), operador de conversión, 53  
\*, 43  
-, 44  
--, 43  
., 25  
.Net, xvii, 2  
/\*, 33  
/, 44  
//, 33  
:, 198, 403  
;, *vea* punto y coma  
@, 277  
[], 232, 242, 262  
\n, 277, 315  
\r, 277, 315  
{}, 114  
||, 125, 150  
+, 43, 44, 47, 276  
++, 43, 144  
<, 118, 145, 237  
<=, 118, 145  
=, 42  
==, 114, 118, 145, 276  
>, 118, 145, 237, 344

>=, 118, 145  
>>, 345  
  
**abstract**, palabra clave, 205  
abstractas, clases, 205, 409, 415  
abstracto, tipo de datos, 190  
alcance, 95, 189  
    de clase, 95  
and, operador, 123, 150  
anidamiento de ciclos, 153, 382  
    de instrucciones if, 126, 380  
apuntador, 77  
archivos, 312-35  
    de procesamiento por lotes, 346  
argumento, 29, 59-91, 62, 63, 64, 76  
    de línea de comandos, 342  
    invocación por referencia, 76  
    invocación por valor, 76  
asignación, 42  
**AutoSize**, 13  
avanzar paso a paso por el código, 396  
ayuda 8, 21  
  
**BackColor**, propiedad, 27  
Bandeja de componentes, 106  
barra  
    de estado, 426  
    de herramientas, 428  
    de progreso, 425  
    de seguimiento, 98, 429  
**base**, palabra clave, 202  
**bool**, palabra clave, 133



- booleanas, variables, 131
- botón de opción, 426
- break**, palabra clave, 128
- búsqueda
  - detallada, 236, 251, 318
  - rápida, 233, 250
- Button**, control, 15, 420
- C#, xvii, 1
- C# 2008 Express Edition, 7
- cadena(s) de texto, 47, 275-295
  - CompareTo**, método, 280
  - comparación de, 279
  - EndsWith**, método, 285
  - IndexOf**, método, 283
  - Insert**, método, 281
  - IsMatch**, método, 286
  - LastIndexOf**, propiedad, 284
  - Length**, propiedad, 282
  - Remove**, método, 281
  - Split**, método, 284
  - StartsWith**, método, 284
  - Substring**, método, 282
  - ToLower**, método, 280
  - ToUpper**, método, 281
  - Trim**, método, 281
- cálculos, 37-58, 211-227
- calendario mensual, 424
- case**, palabra clave, 128
- casilla de verificación, 420
- catch**, palabra clave, 299
- cd, 339
- char**, palabra clave, 278
- clase, 379
- clases, escritura de, 173-195
- Close**, método, 315
- color, 31
- comentarios, 33, 377
- composición, 365
- concatenación, 47
- conflicto de nombres, 69
- Console**, clase, 338
- constantes, 215, 246, 266, 378
- constructor, 96, 183, 203
- Consulta a Frasier, programa, 289
- control, 9, 13
  - con fichas, 427
  - numérico ascendente/descendente, 424
- conversión, 49, 52
  - de tipos, 53, 217, 412
- Convert**, clase, 49
- coordenada, 27
- CreateGraphics**, método, 28, 29
- CreateText**, método, 314
- cuadro
  - combinado, 421
  - de diálogo para manejo de archivos, 325
  - de grupo, 422
  - de herramientas, 8, 11, 13, 418
  - de imagen, 26, 425
  - de lista, 228-239, 423
  - de mensaje, 20, 323
  - de texto, 50, 427
  - de texto enriquecido, 426
- cuerpo de un método, 62
- default**, palabra clave, 130
- depuración, 162-172
- depurador, 162
- desarrollo incremental, 397
- desbordamiento, 222
- diagrama
  - de acción, 117
  - de clases, 201, 354, 359, 410
- dir, 339
- Directory**, clase, 330
- diseño orientado a objetos, 351-374
- distribución, 376
- dividir, 44
- do, instrucción, 152
- do**, palabra clave, 152
- documentación, 386
- double**, palabra clave, 38
- DrawEllipse**, método, 30
- DrawImage**, método, 31
- DrawLine**, método, 30
- DrawRectangle**, método, 25, 29, 30
- editor, 19
- ejecución, 12
- Elegir**, programa, 342
- else**, palabra clave, 116
- encapsulamiento, 177, 380
- entero, 38
- entorno de desarrollo integrado *vea* IDE
- entrada, operaciones de entrada con archivos, 316
- error (*bug*), 163
  - de compilación, 168
  - de conversión, 168
- errores, 18
- es un, 366
- espacios de nombres, 101
- especificación, 389
- estilo, 375
- estructura de datos, 228, 240
- etiqueta, 9, 50, 423
- etiqueta con vínculo, 423
- eventos, 15

excepción, atrapar una, 298  
excepciones, 222, 296-311, 321  
expresiones, 54  
expresiones regulares, 286

**false**, palabra clave, 131  
ficha, 16  
**FileNotFoundException**, 321  
**FillEllipse**, método, 31  
**FillRectangle**, método, 31  
**finally**, palabra clave, 307  
flujos, 313  
for, instrucción, 148  
**for**, palabra clave, 148  
**foreach**, palabra clave, 234, 411  
formato, aplicación de, 19, 212  
formulario, 11, 96  
funciones y constantes matemáticas, 214

genérico, 237  
get, 102, 179  
**get**, palabra clave, 180  
**GetDirectories**, método, 331  
**GetFiles**, método, 331  
Globo, 174  
gráficas, 219  
gráficos, 24-36  
**Graphics**, clase, 28, 430

herencia, 196-210, 365, 370, 371  
historia, 1  
Hola, mundo, programa, 9

IDE, 7, 162  
if, instrucción, 114  
**if**, palabra clave, 114  
if...else, 116  
incremento, 43  
**IndexOutOfRangeException**, 170, 255, 270  
índice, 170, 230, 242, 263, 277  
inicializar una matriz, 247, 267  
iniciar excepción, 298, 306  
**InitializeComponent**, 97  
inspección, 395  
**int**, palabra clave, 38  
**interface**, palabra clave, 403  
interfases, 402-407  
    e interoperabilidad, 405  
    y diseño, 402  
Invasor del ciberespacio, programa, 358  
invocación, 59, 62  
invocar, 59  
**is**, palabra clave, 413  
iteración, 218

Java, 2

**Length**, propiedad, 245  
Lenguaje Unificado de Modelado *vea* UML  
lineamientos de diseño de clases, 370  
lista, 229, 410, 429  
    **Add**, método, 230  
    **Count**, propiedad, 229  
    índice, 230  
    **Insert**, método, 233  
    **Items**, propiedad, 229  
    **RemoveAt**, método, 233  
Lista de errores, ventana, 18

**Main**, 337  
Mandelbrot, 226  
manejo de eventos, 70  
matrices, 240-261, 262-274  
    declaración de, 242, 263  
    de objetos, 252  
    índice de, 242, 263  
    inicialización de, 247, 267  
    **Length**, propiedad de, 245  
    uso como parámetros, 245, 265  
mayúsculas, uso de, 39  
menú, 326  
    principal, 424  
método, 3, 15, 16, 59-91, 60, 182  
Microsoft, 2  
miembros de una clase, 102  
multiplicar, 43, 44

**namespace**, palabra clave, 94  
**new**, palabra clave, 28, 104  
nombres, 39, 376, 380  
not, operador, 123, 150  
nueva línea, 277  
**NullReferenceException**, 109, 171, 190  
números  
    aleatorios, 102  
    de punto flotante, 38  
    reales, 38

objetos, 92-112, 352  
ocultamiento de información, 177, 412  
**OpenText**, método, 314, 317  
operadores  
    aritméticos, 44  
    de comparación, 118, 145  
    lógicos, 123, 150  
or, operador, 123, 150  
**out**, palabra clave, 77  
**OverflowException**, 169  
**override**, palabra clave, 198

- palabras clave, 432
- panel, 424
- parámetro, 29, 62, 63, 64
  - actual, 65
  - formal, 65
- paso a paso por instrucciones, 166
- Pen**, clase, 28
- polimorfismo, 408-416
- private**, palabra clave, 62, 95, 176, 185
- programa, 2
  - de consola, 336-350
- propiedad, 10, 13, 14, 18, 25, 102, 179, 182
- propiedades, 11, 14
- protected**, palabra clave, 197, 198
- proyecto, 7, 9, 17
- prueba, 388-401
  - de la caja blanca, 393
  - de la caja negra, 390
  - estructural, 393
  - exhaustiva, 390
  - funcional, 390
- public**, palabra clave, 177
- punto de interrupción, 164
- punto y coma, 29
  
- Random**, clase, 102, 122, 431
- ReadLine**, método, 313, 317, 338
- ReadToEnd**, método, 317
- recorrido, 395
- redefinición, 200
- redirección de la salida, 344
- Reemplazar**, método, 288
- ref**, palabra clave, 77, 80, 81
- refactorización, 371
- referencia, invocación por, 76, 80, 81
- repetición, 143-61
- restar, 44
- return**, palabra clave, 71
- reutilización, 365
  
- sangría, 19, 115, 377
- secuencia, 33
- secuencias de comandos, 244, 346
- selección, 113-42
- selector de fecha y hora, 422
- set, 102, 179
- set**, palabra clave, 180
  
- Show**, método, 21
- sitio Web, xx
- sobrecarga, 83
- static**, palabra clave, 187
- StreamReader**, clase, 313
- StreamWriter**, clase, 313
- string**, palabra clave, 47
- subíndice, 241
- sugerencia contextual, 429
- switch, instrucción, 127
- switch**, palabra clave, 127
  
- temporizador, 106, 428
- Text**, propiedad, 15, 50
- this**, palabra clave, 82
- throw**, palabra clave, 306
- tiene un, 366
- ToDouble**, método, 49, 276
- ToInt32**, método, 49, 276
- ToString**, método, 276
- true**, palabra clave, 131
- try**, palabra clave, 299
  
- UML, xix
- using**, palabra clave, 94, 101, 314
  
- valor, invocación por, 76
- variables, 37-58
  - a nivel de clase, 95, 176
  - declaración de, 39
  - de instancia, 93, 95, 176
  - locales, 68
- ventana de inspección, 165
- verificación formal, 396
- vinculación
  - dinámica, 414
  - postergada, 414
  - retrasada, 414
- virtual**, palabra clave, 197
- Visual Basic, 2
- Visual Studio, xviii, 7
- void**, palabra clave, 62
  
- while**, instrucción, 144
- while**, palabra clave, 144
- WriteLine**, método, 314, 338





# Visual C# 2008

Este programa puede descargarlo gratuitamente desde el sitio Web de Microsoft Corporation.  
Para llegar a él, por favor siga el vínculo que se encuentra en el sitio Web de *C# para estudiantes*,  
en [www.pearsoneducacion.net/bell](http://www.pearsoneducacion.net/bell).









