3 Introducción a las aplicaciones de C#

¿Qué hay en un nombre? Lo que llamamos rosa aun con otro nombre mantendría su perfume.

—William Shakespeare

Cuando me topo con una decisión, siempre pregunto, "¿Qué sería lo más divertido?"

—Peggy Walker

"Toma un poco más de té", dijo el conejo blanco a Alicia, con gran seriedad. "No he tomado nada todavía." Alicia contestó en tono ofendido, "Entonces no puedo tomar más". "Querrás decir que no puedes tomar menos", dijo el sombrerero loco, "es muy fácil tomar más que nada".

—Lewis Carroll

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Cómo escribir aplicaciones simples en C# usando código, en lugar de la programación visual.
- Escribir instrucciones que envíen y reciban datos hacia/desde la pantalla.
- Cómo declarar y utilizar datos de diversos tipos.
- Almacenar y recuperar datos de la memoria.
- Utilizar los operadores aritméticos.
- Determinar el orden en que se aplican los operadores.
- Escribir instrucciones de toma de decisiones.
- Utilizar los operadores relacionales y de igualdad.
- Utilizar cuadros de diálogo de mensaje para mostrar mensajes.

Plan general

3.1 Introducción

Ahora presentaremos la programación de aplicaciones en C#, que nos facilita un método disciplinado para su diseño. La mayoría de las aplicaciones de C# que estudiaremos en este libro procesa información y muestra resultados. En este capítulo introduciremos las ***aplicaciones de consola***; éstas envían y reciben texto en una ***ventana de consola***. En Microsoft Windows 95/98/ME, la ventana de consola es el ***símbolo de MS-DOS***. En otras versiones de Microsoft Windows, la ventana de consola es el ***Símbolo del sistema***.

Empezaremos con diversos ejemplos que sólo muestran mensajes en la pantalla. Después demostraremos una aplicación que obtiene dos números de un usuario, calcula la suma y muestra el resultado. Aprenderá a realizar diversos cálculos aritméticos y a guardar los resultados para utilizarlos posteriormente. Muchas aplicaciones contienen lógica que requiere que la aplicación tome decisiones; el último ejemplo de este capítulo demuestra los fundamentos de la toma de decisiones al enseñarle cómo comparar números y mostrar mensajes con base en los resultados de las comparaciones. Por ejemplo, la aplicación muestra un mensaje que indica que dos números son iguales sólo si tienen el mismo valor. Analizaremos cuidadosamente cada ejemplo, una línea a la vez.

3.2 Una aplicación simple en C#: mostrar una línea de texto

Consideremos ahora una aplicación simple que muestra una línea de texto. (Más adelante en esta sección veremos cómo compilar y ejecutar una aplicación.) La figura 3.1 muestra esta aplicación y su salida. La aplicación ilustra varias características importantes del lenguaje C#. Para su conveniencia, cada uno de los programas que presentamos en este libro incluye números de línea, que no forman parte del verdadero código de C#. En la sección 3.3 le indicaremos cómo mostrar los números de línea para su código de C# en el IDE. Pronto veremos que la línea 10 es la que hace el verdadero trabajo de la aplicación; a saber, muestra en la pantalla la frase `¡Bienvenido a la programación en C#!` Ahora analizaremos cada una de las líneas de esta aplicación.

La línea 1

```
// Figura 3.1: Bienvenido1.cs
```

comienza con //, lo cual indica que el resto de la línea es un ***comentario***. Empezamos cada aplicación con un comentario en el que se indica el número de figura y el nombre del archivo en el que se guarda la aplicación.

A un comentario que comienza con // se le llama ***comentario de una sola línea***, ya que termina al final de la línea en la que aparece. Este tipo de comentario también puede comenzar en medio de una línea y continuar hasta el final de la misma (como en las líneas 7, 11 y 12).

Los ***comentarios delimitados*** como

```
/* Éste es un comentario
   delimitado. Puede dividirse
   en muchas líneas*/
```

```
1  // Fig. 3.1: Bienvenido1.cs
2  // Aplicación para imprimir texto.
3  using System;
4
5  public class Bienvenido1
6  {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10       Console.WriteLine( "¡Bienvenido a la programación en C#!" );
11    } // fin del método Main
12 } // fin de la clase Bienvenido1
```

```
¡Bienvenido a la programación en C#!
```

Figura 3.1 | Aplicación para imprimir texto en pantalla.

pueden esparcirse a través de varias líneas. Este tipo de comentario comienza con el delimitador /* y termina con el delimitador */. El compilador ignora todo el texto entre los delimitadores. C# incorporó los comentarios delimitados y los de una sola línea de los lenguajes de programación C y C++, en forma respectiva. En este libro utilizaremos comentarios de una sola línea en nuestros programas.

La línea 2

```
// Aplicación para imprimir texto.
```

es un comentario de una sola línea, que describe el propósito de la aplicación.

La línea 3

```
using System;
```

es una ***directiva using***, que ayuda al compilador a localizar una clase que se utiliza en esta aplicación. Uno de los puntos fuertes de C# es su robusto conjunto de clases predefinidas que usted puede utilizar, en lugar de "reinventar la rueda". Estas clases se organizan bajo ***espacios de nombres***: colecciones con nombre de clases relacionadas. A los espacios de nombres de .NET se les conoce como ***Biblioteca de Clases del .NET Framework*** **(*FCL*)**. Cada directiva `using` identifica las clases predefinidas que puede utilizar una aplicación de C#. La directiva `using` en la línea 3 indica que este ejemplo utiliza clases del espacio de nombres `System`, el cual contiene la clase predefinida `Console` (que veremos en breve) que se utiliza en la línea 10, y muchas otras clases útiles.

Error común de programación 3.1

Todas las directivas `using` deben aparecer antes de cualquier otro código (excepto los comentarios) en un archivo de código fuente de C#; en caso contrario se produce un error de compilación.

Tip de prevención de errores 3.1

Si olvida incluir una directiva `using` para una clase que se utilice en su aplicación, es muy probable que se produzca un error de compilación en el que se muestre un mensaje como `"El nombre 'Console' no existe en el contexto actual."` Cuando esto ocurra, verifique que haya proporcionado las directivas `using` apropiadas y que los nombres en las directivas `using` estén deletreados de forma correcta, incluyendo el uso apropiado de letras mayúsculas y minúsculas.

Para cada nueva clase .NET que utilicemos, debemos indicar el espacio de nombres en el que se encuentra. Esta información es importante, ya que nos ayuda a localizar las descripciones de cada clase en la ***documentación de .NET***. Puede encontrar una versión basada en Web de este documento en las siguientes páginas Web:

```
msdn2.microsoft.com/en-us/library/ms229335  (inglés)
msdn2.microsoft.com/es-es/library/ms306608 (español)
```

También puede encontrar esta información en la documentación de Visual C# Express, en el menú **Ayuda**. Incluso puede colocar el cursor sobre el nombre de cualquier clase o método .NET y después oprimir *F1* para obtener más información.

La línea 4 es tan sólo una línea en blanco. Las líneas en blanco, los caracteres de espacio y los caracteres de tabulación se conocen como ***espacio en blanco***. (En específico, los caracteres de espacio y los tabuladores se conocen como ***caracteres de espacio en blanco***.) El compilador ignora todo el espacio en blanco. En este capítulo y en algunos de los siguientes hablaremos sobre las convenciones para el uso de espacio en blanco, con el fin mejorar la legibilidad de las aplicaciones.

La línea 5

```
public class Bienvenido1
```

comienza una ***declaración de clase*** para la clase `Bienvenido1`. Toda aplicación consiste de cuando menos una declaración de clase, la cual usted (como programador) define. A estas clases se les conoce como ***clases definidas por el usuario***. La ***palabra clave*** `class` introduce una declaración de clase y va seguida inmediatamente del ***nombre de la clase*** (`Bienvenido1`). Las palabras clave (conocidas como ***palabras reservadas***) están reservadas para uso exclusivo del lenguaje C# y siempre se escriben en minúsculas. En la figura 3.2 se muestra la lista completa de palabras clave de C#.

Por convención, todos los nombres de las clases comienzan con mayúscula y la primera letra de cada palabra que incluyen también se escribe en mayúscula (por ejemplo, `NombreClaseEjemplo`). Por lo general, a esto se le conoce como ***estilo de mayúsculas/minúsculas Pascal***. El nombre de una clase es un ***identificador***, una serie de caracteres que consiste de letras, dígitos y guiones bajos (`_`), que no comienza con un dígito y que no contiene espacios. Algunos identificadores válidos son `Bienvenido1`, `identificador`, `_valor` y `m_campoEntrada1`. El nombre `7boton` no es un identificador válido, ya que comienza con un dígito; el nombre `campo entrada` tampoco lo es, ya que contiene un espacio. Por lo general, un identificador que no comienza con letra mayúscula no es el nombre de una clase. C# es ***sensible a mayúsculas y minúsculas***, es decir, las letras mayúsculas y minúsculas son distintas, por lo que `a1` y `A1` son identificadores distintos (pero ambos son válidos). También se puede colocar el carácter `@` al principio de un identificador. Esto indica que una palabra debe interpretarse como identificador, aun y cuando sea una palabra clave (por ejemplo, `@int`). De esta forma el código de C# puede utilizar código escrito en otros lenguajes .NET, en los que un identificador podría tener el mismo nombre que una palabra clave de C#.

Palabras clave de C#

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

Figura 3.2 | Palabras clave de C#.

Buena práctica de programación 3.1

Por convención, siempre se debe comenzar el identificador del nombre de una clase con letra mayúscula y cada palabra subsiguiente en el identificador se debe comenzar con letra mayúscula.

Error común de programación 3.2

C# es sensible a mayúsculas y minúsculas. Por lo general, si no se utiliza la combinación apropiada de letras mayúsculas y minúsculas para un identificador, se produce un error de compilación.

En los capítulos del 3 al 8, cada una de las clases que definimos comienza con la palabra clave ***`public`***. Por ahora sólo requeriremos esta palabra clave. En el capítulo 9 aprenderá más acerca de las clases `public` y `no public`. Cuando guarda su declaración de clase `public` en un archivo, por lo general el nombre del archivo es el de la clase, seguido de la extensión de nombre de archivo `.cs`. Para nuestra aplicación, el nombre de archivo es `Bienvenido1.cs`.

Buena práctica de programación 3.2

Por convención, un archivo que contiene una sola clase `public` *debe tener un nombre que sea idéntico al nombre de la clase (más la extensión* `.cs`*) en términos tanto de ortografía como de uso de mayúsculas y minúsculas. Si nombra sus archivos de esta forma facilitará a los demás programadores (y a usted) el proceso de determinar la ubicación de las clases de una aplicación.*

Una ***llave izquierda*** (en la línea 6 de la figura 3.1), `{`, comienza el cuerpo de cualquier declaración de clase. Su correspondiente ***llave derecha*** (en la línea 12), `}`, debe terminar cada declaración de clase. Observe que las líneas de la 7 a la 11 tienen sangría. Esta sangría es una de las convenciones de espaciado que mencionamos antes. Definiremos cada convención de espaciado como una Buena práctica de programación.

Tip de prevención de errores 3.2

Siempre que escriba una llave izquierda de apertura, `{`*, en su aplicación, escriba de inmediato la llave derecha de cierre,* `}`*, y después reposicione el cursor entre las llaves y utilice sangría para comenzar a escribir el cuerpo. Esta práctica le ayudará a evitar errores por la omisión de llaves.*

Buena práctica de programación 3.3

Aplique sangría a todo el cuerpo de cada declaración de clase; utilice un "nivel" de sangría entre las llaves izquierda y derecha que delimiten el cuerpo de su clase. Este formato enfatiza la estructura de la declaración de la clase y mejora la legibilidad. Para dejar que el IDE aplique formato a su código, seleccione **Edición > Avanzado > Dar formato al documento**.

Buena práctica de programación 3.4

Establezca una convención para el tamaño de sangría que prefiera y después aplique esa convención de manera uniforme. Puede utilizar la tecla ***Tab*** *para crear sangrías, pero los marcadores de tabulación varían entre los distintos editores de texto. Le recomendamos que utilice tres espacios para formar cada nivel de sangría. En la sección 3.3 le mostraremos cómo hacer esto.*

Error común de programación 3.3

Si las llaves no se escriben en pares se produce un error de sintaxis.

La línea 7

```
// El método Main comienza la ejecución de la aplicación de C#
```

es un comentario que indica el propósito de las líneas 8-11 de la aplicación. La línea 8

```
public static void Main( string[] args )
```

es el punto de inicio de toda aplicación. Los ***paréntesis*** después del identificador `Main` indican que es un bloque de construcción de aplicaciones llamado método. Por lo general, las declaraciones de clases contienen uno o más métodos. Sus nombres siguen casi siempre las convenciones de estilo de mayúsculas/minúsculas Pascal que se utilizan para los nombres de las clases. Para cada aplicación es obligatorio que uno de los métodos en una clase se llame `Main` (que por lo general se define como se muestra en la línea 8); de no ser así, la aplicación no se ejecutará. Los métodos pueden realizar tareas y devolver información cuando completan su trabajo. La palabra clave ***`void`*** (línea 8) indica que este método no devolverá información después de completar su tarea. Más adelante veremos que muchos métodos sí devuelven información; en los capítulos 4 y 7 aprenderá más acerca de ellos. Por ahora, sólo imite la primera línea de `Main` en sus aplicaciones.

La llave izquierda en la línea 9 comienza el ***cuerpo de la declaración del método***. El cuerpo del método debe terminar con su correspondiente llave derecha (línea 11 de la figura 3.1). Observe que la línea 10 en el cuerpo del método tiene sangría entre las llaves.

Buena práctica de programación 3.5

Al igual que con las declaraciones de clases, debe aplicar sangría a todo el cuerpo de la declaración de cada método; para ello utilice un "nivel" de sangría entre las llaves izquierda y derecha que definen el cuerpo del método. Este formato hace que la estructura del método resalte y mejora su legibilidad.

La línea 10

```
Console.WriteLine( "¡Bienvenido a la programación en C#!" );
```

instruye a la computadora para que ***realice una acción***; a saber, imprimir (es decir, mostrar en pantalla) la ***cadena*** de caracteres contenida entre los dos símbolos de comillas dobles. A una cadena también se le conoce como ***cadena de caracteres***, un ***mensaje*** o una ***literal de cadena***. A los caracteres entre símbolos de comillas dobles les llamaremos simplemente ***cadenas***. El compilador no ignora los caracteres de espacio en blanco en las cadenas.

La clase ***`Console`*** proporciona características de ***entrada/salida estándar***, las cuales permiten a las aplicaciones leer y mostrar texto en la ventana de consola desde la cual se ejecuta la aplicación. El ***método `Console.WriteLine`*** muestra (o imprime) una línea de texto en la ventana de consola. La cadena entre paréntesis en la línea 10 es el ***argumento*** para el método. La tarea del método `Console.WriteLine` es mostrar (o enviar de salida) su argumento en la ventana de consola. Cuando `Console.WriteLine` completa su tarea, posiciona el ***cursor de la pantalla*** (el símbolo destellante que indica en dónde se mostrará el siguiente carácter) al principio de la siguiente línea en la ventana de consola. (Este movimiento del cursor es similar a lo que ocurre cuando un usuario oprime *Intro* mientras escribe en un editor de texto: el cursor se desplaza al principio de la siguiente línea en el archivo.)

A toda la línea 10, incluyendo `Console.WriteLine`, los paréntesis, el argumento `"¡Bienvenido a la programación en C#!"` entre paréntesis y el ***punto y coma (`;`)***, se le conoce como ***instrucción***. Cada instrucción termina con un punto y coma. Cuando se ejecuta la instrucción de la línea 10, muestra el mensaje `¡Bienvenido a la programación en C#!` en la ventana de consola. Por lo general, un método está compuesto de una o más instrucciones, que realizan la tarea de ese método.

Error común de programación 3.4

Si se omite el punto y coma al final de una instrucción se produce un error de sintaxis.

Algunos programadores tienen problemas para asociar las llaves izquierda y derecha (`{` y `}`) que delimitan el cuerpo de la declaración de una clase o de un método cuando leen o escriben una aplicación. Por esta razón, es común incluir un comentario después de cada llave derecha de cierre (`}`) que termina la declaración de un método, y después de cada llave derecha de cierre que termina la declaración de una clase. Por ejemplo, la línea 11

```
} // fin del método Main
```

especifica la llave derecha de cierre del método `Main`, y la línea 12

```
} // fin de la clase Bienvenido1
```

especifica la llave derecha de cierre de la clase `Bienvenido1`. Cada uno de estos comentarios indica el método o la clase que termina con esa llave derecha. Visual Studio puede ayudarle a localizar las llaves que concuerden en su código. Sólo tiene que colocar el cursor a un lado de una llave y Visual Studio resaltará la otra.

Buena práctica de programación 3.6

Si después de la llave derecha de cierre del cuerpo de un método o de la declaración de una clase se coloca un comentario que indique el método o la declaración de la clase a la cual pertenece esa llave, se mejora la legibilidad de la aplicación.

3.3 Cómo crear una aplicación simple en Visual C# Express

Ahora que le hemos presentado nuestra primera aplicación de consola (figura 3.1), veremos una explicación paso a paso de cómo compilarla y ejecutarla mediante el uso de Visual C# Express.

Creación de la aplicación de consola

Después de abrir Visual C# 2005 Express, seleccione **Archivo > Nuevo proyecto...** para que aparezca el cuadro de diálogo **Nuevo proyecto** (figura 3.3) y luego seleccione la plantilla **Aplicación de consola**. En el campo **Nombre** del cuadro de diálogo, escriba `Bienvenido1` y haga clic en **Aceptar** para crear el proyecto. Ahora el IDE debe contener la aplicación de consola, como se muestra en la figura 3.4. Observe que la ventana del editor ya contiene algo de código proporcionado por el IDE. Parte de este código es similar al de la figura 3.1. Otra parte no lo es, ya que utiliza características que no hemos visto todavía. El IDE inserta este código adicional para ayudar a organizar la aplicación y proporcionar acceso a ciertas clases comunes en la Biblioteca de clases del .NET Framework; en este punto del libro, este código no se requiere ni es relevante a la discusión de esta aplicación. Por lo tanto, puede eliminarlo.

El esquema de colores del código que utiliza el IDE se llama ***resaltado de sintaxis por colores***; este esquema nos ayuda a diferenciar en forma visual los elementos de la aplicación. Las palabras clave aparecen en color azul y el resto del texto en color negro. Cuando hay comentarios, aparecen en color verde. En este libro sombrearemos nuestro código de manera similar; texto en negrita y cursiva para las palabras clave, cursiva para los comentarios, negrita color gris para las literales y constantes, y color negro para el resto del texto. Un ejemplo de literal es la cadena que se pasa a `Console.WriteLine` en la línea 10 de la figura 3.1. Para personalizar los colores que se muestran en el editor de código seleccione **Herramientas > Opciones...** A continuación aparecerá el cuadro de diálogo

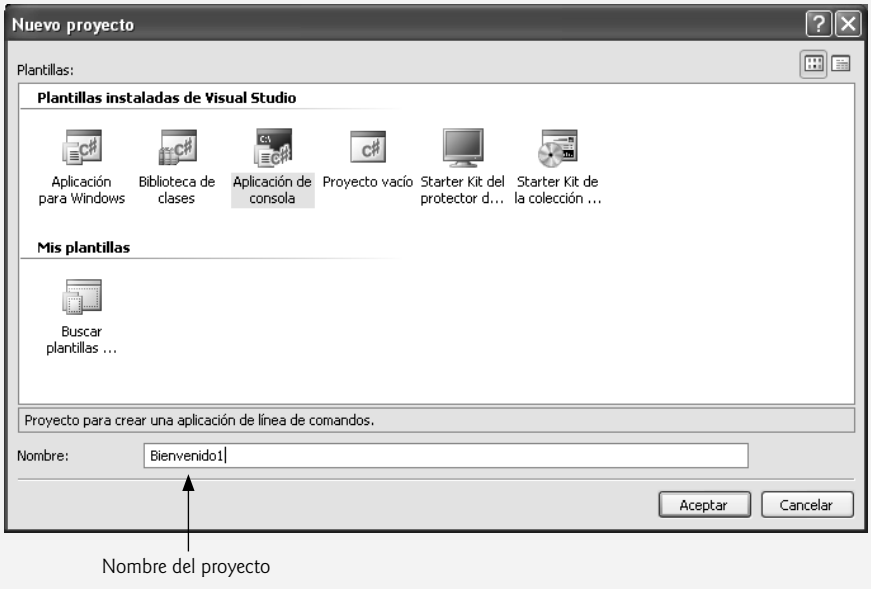

Figura 3.3 | Creación de una **Aplicación de consola** con el cuadro de diálogo **Nuevo proyecto**.

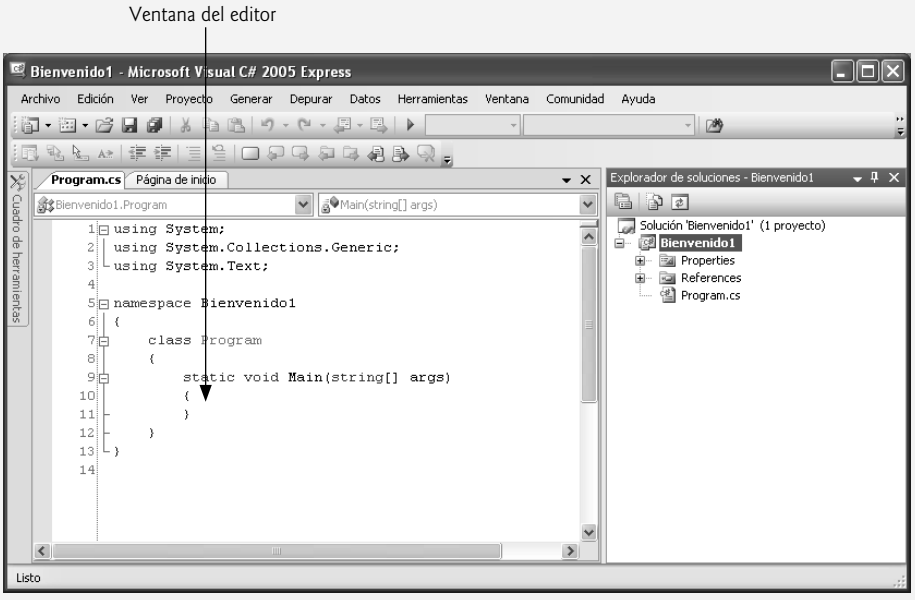

Figura 3.4 | El IDE con una aplicación de consola abierta.

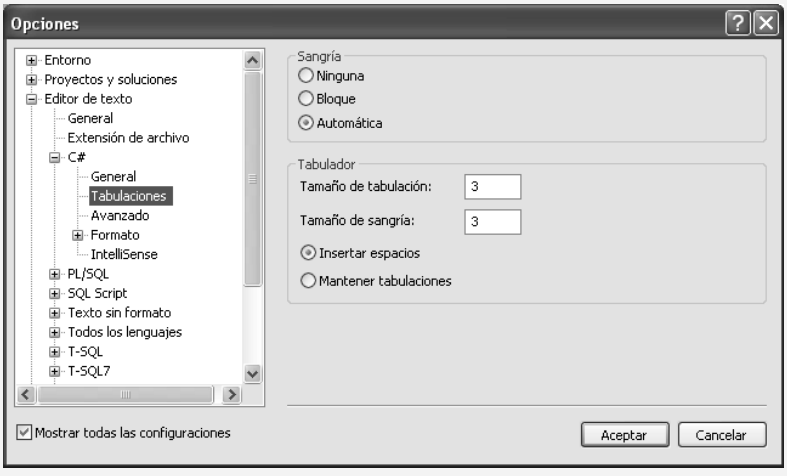

Figura 3.5 | Modificación de las opciones de configuración del IDE.

Opciones (figura 3.5). Después haga clic en el signo más (+) que está a un lado de **Entorno** y seleccione **Fuentes y colores.** Aquí puede modificar los colores para varios elementos de código.

Modificación de la configuración del editor para mostrar números de línea

Visual C# Express le ofrece muchas formas para personalizar su manera de trabajar con el código. En este paso modificará las configuraciones para que su código concuerde con el de este libro. Para que el IDE muestre los números de línea, seleccione **Herramientas > Opciones...** En el cuadro de diálogo que se despliega, haga clic en la casilla de verificación **Mostrar todas las configuraciones** que se encuentra en la parte inferior izquierda, después haga clic en el signo más que está a un lado de **Editor de texto** en el panel izquierdo y seleccione **Todos los lenguajes.** En el panel derecho, active la casilla de verificación **Números de línea.** Mantenga abierto el cuadro de diálogo **Opciones.**

Establecer la sangría del código a tres espacios

En el cuadro de diálogo **Opciones** que abrió en el paso anterior (figura 3.5), haga clic en el signo más que está a un lado de C# en el panel izquierdo y seleccione **Tabulaciones**. Escriba el número **3** en los campos **Tamaño de tabulación** y **Tamaño de sangría**. Todo el nuevo código que agregue utilizará ahora tres espacios para cada nivel de sangría. Haga clic en **Aceptar** para guardar sus nuevas configuraciones, cierre el cuadro de diálogo y regrese a la ventana del editor.

Modificación del nombre del archivo de aplicación

Para las aplicaciones que crearemos en este libro, modificaremos el nombre predeterminado del archivo de aplicación (es decir, `Program.cs`) para ponerle un nombre más descriptivo. Para cambiar el nombre del archivo, haga clic en el nombre `Program.cs`, en la ventana **Explorador de soluciones**. A continuación aparecerán las propiedades del archivo de aplicación en la ventana **Propiedades** (figura 3.6). Cambie la ***propiedad Nombre de archivo*** a `Bienvenido1.cs`.

Escritura de código

En la ventana del editor (figura 3.4), escriba el código de la figura 3.1. Después de escribir (en la línea 10) el nombre de la clase y un punto (por ejemplo, `Console.`) aparecerá una ventana con una barra de desplazamiento (figura 3.7). A esta característica del IDE se le llama ***IntelliSense***; sirve para listar los ***miembros*** de una clase, incluyendo los nombres de los métodos. A medida que usted escribe caracteres, Visual C# Express resalta el primer miembro que concuerda con todos los caracteres escritos y después muestra un cuadro de información de herramienta (tool tip), que contiene una descripción de ese miembro. Puede escribir el nombre del miembro completo (por ejemplo, `WriteLine`), hacer doble clic en el nombre del miembro que se encuentra en la lista u oprimir *Tab* para completar el nombre. Una vez que se proporciona el nombre completo, la ventana *IntelliSense* se cierra.

Cuando escribe el carácter de paréntesis abierto (, después de `Console.WriteLine`, aparece la ventana ***Información de parametros*** (figura 3.8). Esta ventana contiene información sobre los parámetros del método. Como aprenderá en el capítulo 7, puede haber varias versiones de un método; esto es, una clase puede definir varios métodos que tengan el mismo nombre, siempre y cuando tengan distintos números y/o tipos de parámetros.

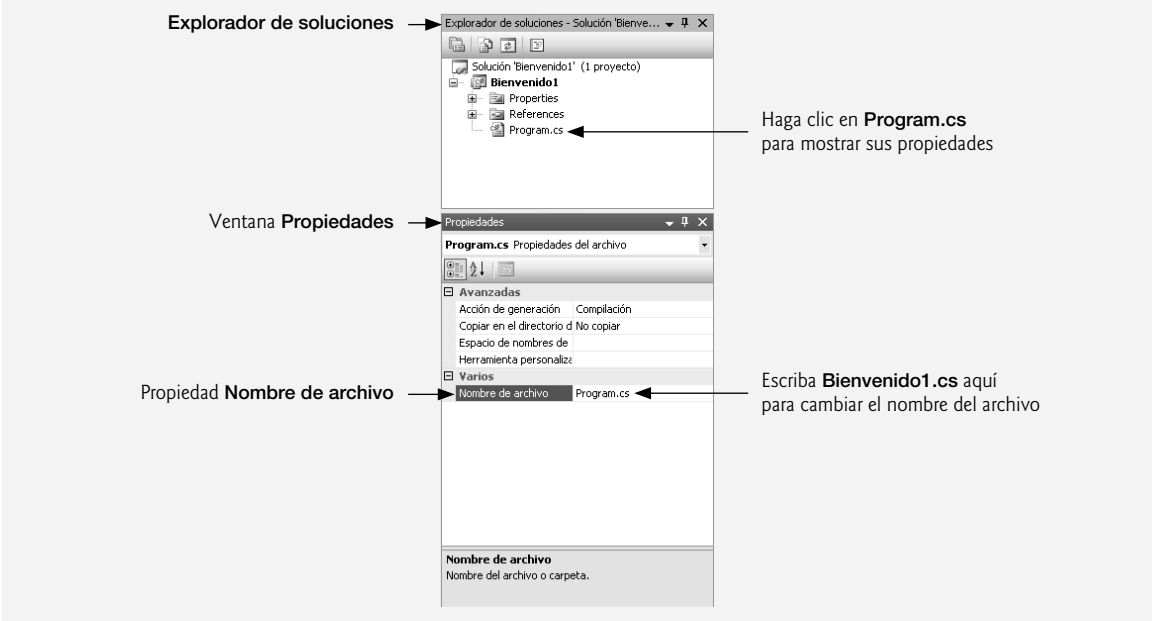

Figura 3.6 | Modificación del nombre del programa en la ventana **Propiedades**.

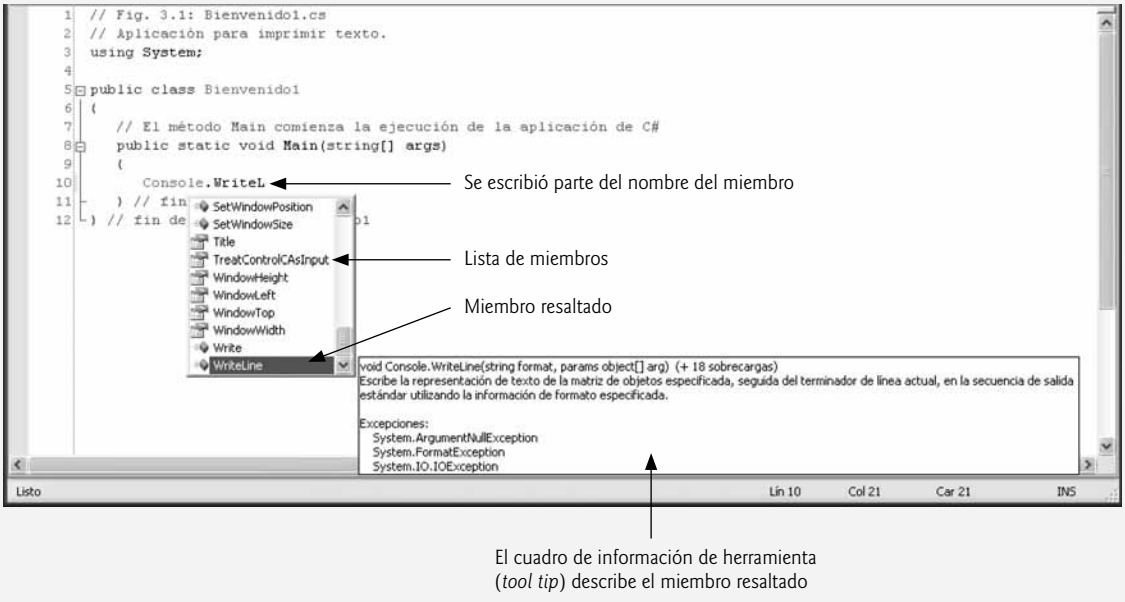

Figura 3.7 | Característica *IntelliSense* de Visual C# Express.

Por lo general, todos estos métodos realizan tareas similares. La ventana *Información de parámetros* indica cuántas versiones del método seleccionado hay disponibles y proporciona flechas hacia arriba y hacia abajo para desplazarse a través de las distintas versiones. Por ejemplo, hay 19 versiones del método `WriteLine`; en nuestra aplicación utilizamos una de esas 19 versiones. La ventana *Información de parámetros* es una de las muchas características que ofrece el IDE para facilitar el desarrollo de aplicaciones. En los siguientes capítulos aprenderá más acerca de la información que se muestra en estas ventanas. La ventana *Información de parámetros* es útil, en especial, cuando deseamos ver las distintas formas en las que se puede utilizar un método. Del código de la figura 3.1 ya sabemos que nuestra intención es mostrar una cadena con `WriteLine`, y como sabemos con exactitud cuál versión de `WriteLine` deseamos utilizar, por el momento sólo hay que cerrar la ventana *Información de parámetros* mediante la tecla *Esc*.

Guardar la aplicación

Seleccione **Archivo > Guardar todo** para que aparezca el cuadro de diálogo **Guardar proyecto** (figura 3.9). En el cuadro de texto **Ubicación**, especifique el directorio en el que desea guardar el proyecto. Nosotros optamos por guardarlo en el directorio `MisProyectos` de la unidad `C:`. Seleccione la casilla de verificación **Crear directorio para la solución** (para que Visual Studio cree el directorio, en caso de que no exista todavía) y haga clic en **Guardar**.

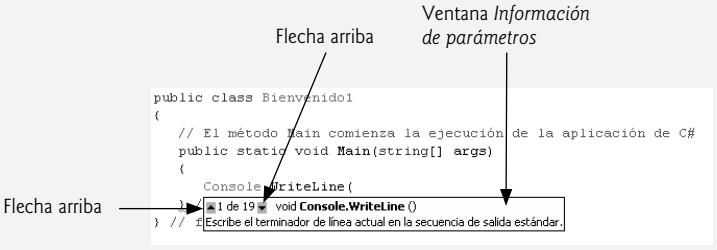

Figura 3.8 | Ventana *Información de parámetros*.

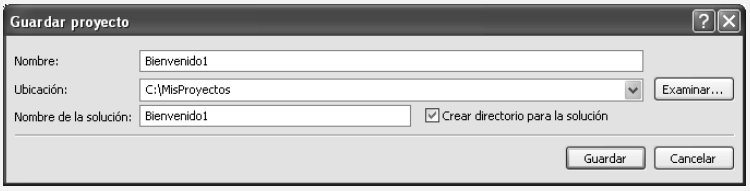

Figura 3.9 | Cuadro de diálogo **Guardar proyecto**.

Compilación y ejecución de la aplicación

Ahora está listo para compilar y ejecutar su aplicación. Dependiendo del tipo de aplicación, el compilador puede compilar el código en archivos con una ***extensión*** `.exe` (***ejecutable***), con una ***extensión*** `.dll` (***biblioteca de vínculos dinámicos***) o con alguna otra disponible. Dichos archivos se llaman ensamblados y son las unidades de empaquetamiento para el código de C# compilado. Estos ensamblados contienen el código de Lenguaje intermedio de Microsoft (MSIL) para la aplicación.

Para compilar la aplicación, seleccione **Generar > Generar solución**. Si la aplicación no contiene errores de sintaxis, su aplicación de consola se compilara en un archivo ejecutable (llamado `Bienvenido1.exe`, en el directorio del proyecto). Para ejecutar esta aplicación de consola (es decir, `Bienvenido1.exe`), seleccione **Depurar > Iniciar sin depurar** (u oprima *<Ctrl> F5*), con lo cual se invocará el método `Main` (figura 3.1). La instrucción en la línea 10 de `Main` muestra `¡Bienvenido a la programación en C#!`. La figura 3.10 muestra los resultados de la ejecución de esta aplicación. Observe que los resultados se despliegan en una ventana de consola. Deje la aplicación abierta en Visual C# Express; más adelante en esta sección volveremos a utilizarla.

Ejecución de la aplicación desde el símbolo del sistema

Como dijimos al principio del capítulo, puede ejecutar aplicaciones fuera del IDE a través del **Símbolo del sistema**. Esto es útil cuando sólo desea ejecutar una aplicación, en vez de abrirla para modificarla. Para abrir el **Símbolo del sistema**, seleccione **Inicio > Todos los programas > Accesorios > Símbolo del sistema**. [*Nota*: Los usuarios de Windows 2000 deben sustituir **Todos los programas** con **Programas**.] La ventana (figura 3.11) muestra la información de derechos de autor, seguida de un símbolo que indica el directorio actual. De manera predeterminada, el símbolo especifica el directorio actual del usuario en el equipo local (en nuestro caso, `C:\Documents and Settings\deitel`). En su equipo, el nombre de carpeta `deitel` se sustituirá con su nombre de usuario. Escriba el comando `cd` (que significa "cambiar directorio"), seguido del modificador `/d` (para cambiar de unidad, en caso de ser necesario) y después el directorio en el que se encuentra el archivo `.exe` de la aplicación (es decir, el directorio `Release` de su aplicación). Por ejemplo, el comando `cd/d C:\MisProyectos\Bienvenido1\Bienvenido1\bin\Release` (figura 3.12) cambia el directorio actual al directorio `Release` de la aplicación `Bienvenido1` en la unidad `C:`. El siguiente símbolo muestra el nuevo directorio. Después de cambiar al directorio actual, usted puede ejecutar la aplicación compilada con sólo escribir el nombre del archivo `.exe` (es decir, `Bienvenido1`). La apli-

Figura 3.10 | La ejecución de la aplicación que se muestra en la figura 3.1.

cación se ejecutará hasta completarse y después aparecerá de nuevo el símbolo del sistema, esperando el siguiente comando. Para cerrar el **Símbolo del sistema**, escriba `exit` (figura 3.12) y oprima *Intro*.

Visual C# 2005 Express mantiene un directorio `Debug` y un directorio `Release` en el directorio `bin` de cada proyecto. El primero contiene una versión de la aplicación que puede utilizarse con el depurador (vea el apéndice C, Uso del depurador de Visual Studio® 2005). El directorio `Release` contiene una versión optimizada que usted puede proporcionar a sus clientes. En la versión completa de Visual Studio 2005 puede seleccionar la versión específica que desea generar en la lista desplegable **Configuraciones de la solución** de las barras de herramientas que se encuentran en la parte superior del IDE. La versión predeterminada es `Debug`.

[*Nota*: muchos entornos muestran ventanas **Símbolo del sistema** con fondos negros y texto blanco. Nosotros ajustamos estas configuraciones en nuestro entorno para mejorar la legibilidad de las capturas de pantalla.]

Errores de sintaxis, mensajes de error y la ventana Lista de errores

Regrese a la aplicación en Visual C# Express. Cuando escribe una línea de código y oprime *Intro*, el IDE responde ya sea mediante la aplicación de resaltado de sintaxis por colores o mediante la generación de un ***error de sintaxis***, el cual indica una violación a las reglas de Visual C# para crear aplicaciones correctas (es decir, una o más instrucciones no están escritas correctamente). Los errores de sintaxis se producen por diversas razones, como omitir paréntesis y palabras clave mal escritas.

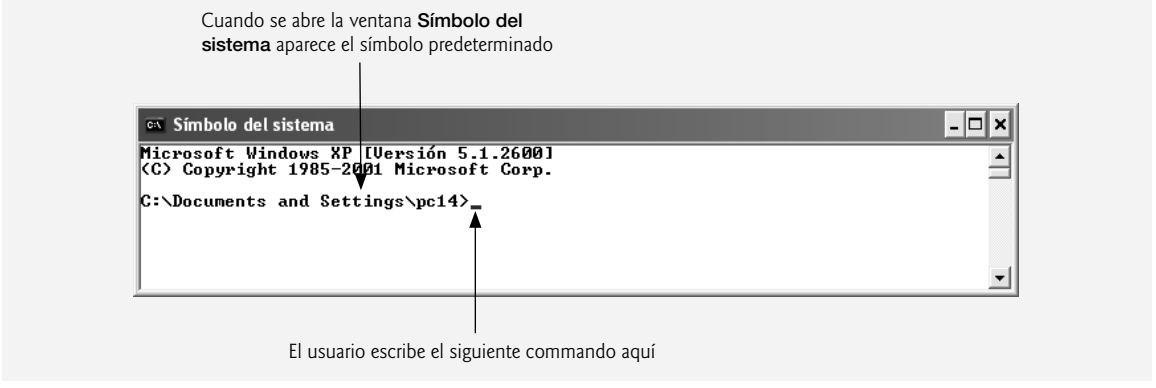

Figura 3.11 | Ejecución de la aplicación que se muestra en la figura 3.1, desde una ventana **Símbolo del sistema**.

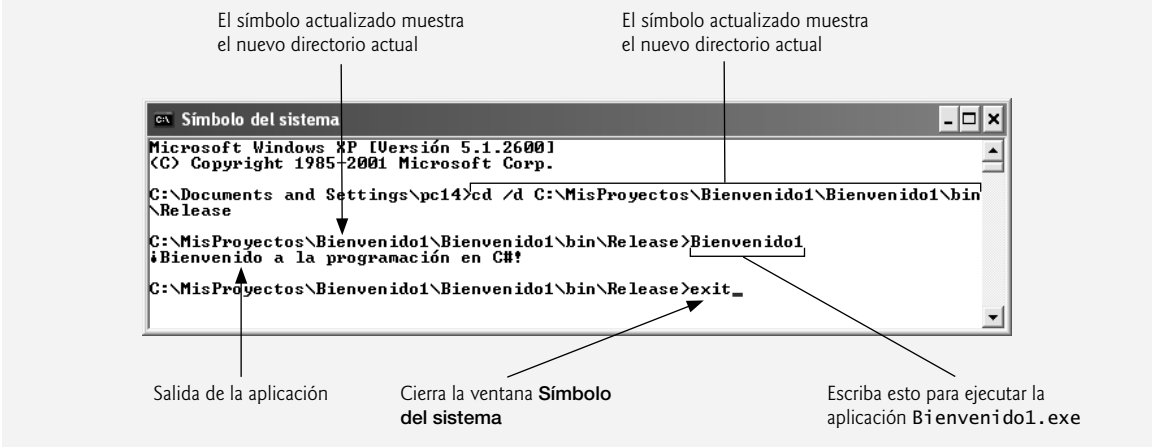

Figura 3.12 | Ejecución de la aplicación que se muestra en la figura 3.1, desde una ventana **Símbolo del sistema**.

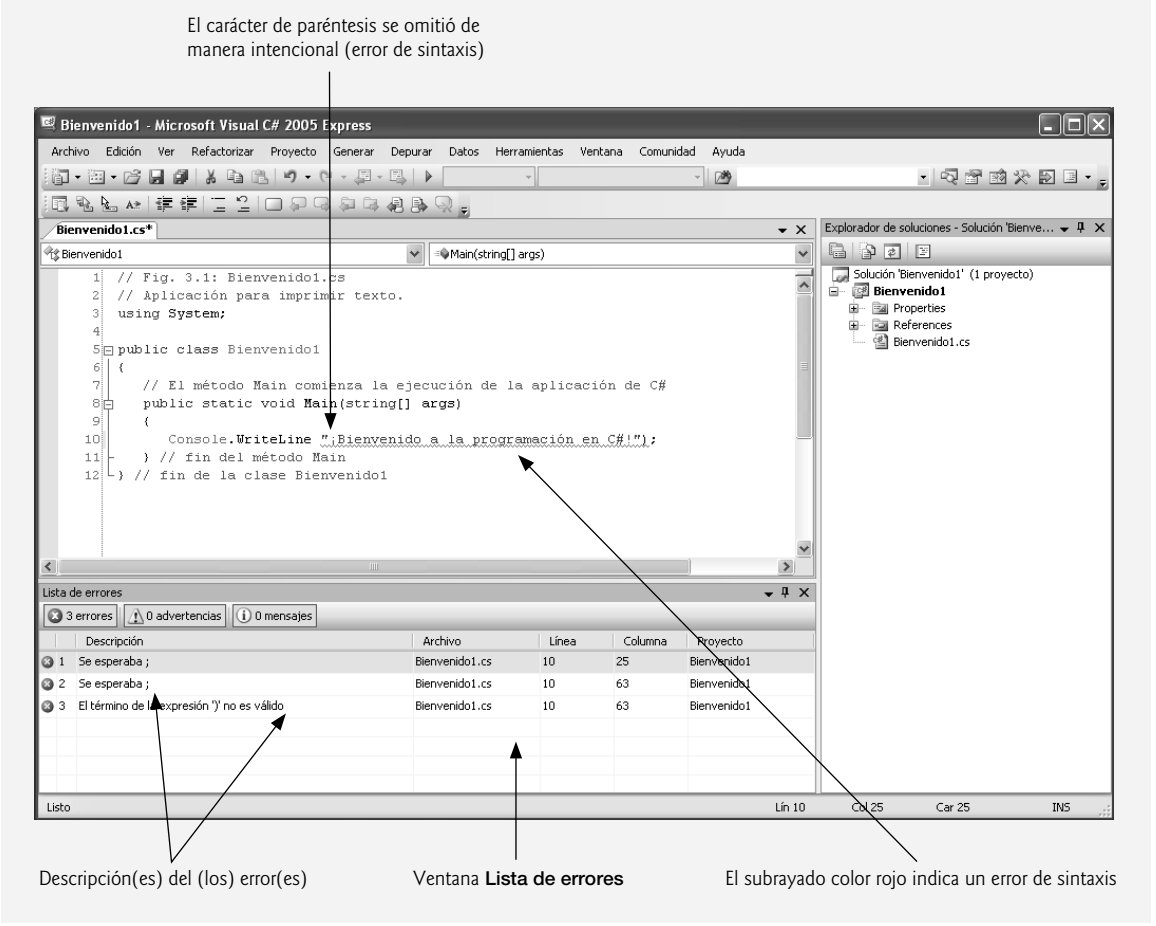

Figura 3.13 | Errores de sintaxis indicados por el IDE.

Cuando se produce un error de sintaxis, el IDE lo subraya en color rojo y proporciona una descripción del error en la ***ventana Lista de errores*** (figura 3.13). Si la ventana no está visible en el IDE, seleccione **Ver > Lista de errores** para mostrarla. En la figura 3.13 omitimos intencionalmente el primer paréntesis en la línea 10. El primer error contiene el texto **"Se esperaba ;"** y especifica que el error está en la columna 25 de la línea 10. Este mensaje de error aparece cuando el compilador piensa que la línea contiene una instrucción completa, seguida de un punto y coma, y el comienzo de otra instrucción. El segundo error contiene el mismo texto, pero especifica que está en la columna 54 de la línea 10, ya que el compilador piensa que éste es el final de la segunda instrucción. El tercer error tiene el texto **"El término de la expresión ')' no es válido"**, ya que el compilador está confundido por el paréntesis derecho sin su correspondiente paréntesis izquierdo. Aunque estamos tratando de incluir sólo una instrucción en la línea 10, el paréntesis izquierdo que falta ocasiona que el compilador asuma de manera incorrecta que hay más de una instrucción en esa línea, que malinterprete el paréntesis derecho y genere *tres* mensajes de error.

Tip de prevención de errores 3.3

Un error de sintaxis puede producir varias entradas en la ventana ***Lista de errores****. Con cada error que usted corrija, podrá eliminar varios mensajes de error subsiguientes al compilar nuevamente su aplicación. Por lo tanto, cuando vea un error que sabe cómo corregir, hágalo y vuelva a compilar el programa; esto puede hacer que desaparezcan varios errores.*

3.4 Modificación de su aplicación simple en C#

En esta sección continuamos con nuestra introducción a la programación en C# con dos ejemplos que modifican el de la figura 3.1, para imprimir texto en una línea mediante el uso de varias instrucciones e imprimir texto en varias líneas mediante el uso de una sola instrucción.

Mostrar una sola línea de texto con varias instrucciones

El texto `"¡Bienvenido a la programación en C#!"` puede mostrarse de varias formas. La clase `Bienvenido2`, que se muestra en la figura 3.14, utiliza dos instrucciones para producir el mismo resultado que el que se muestra en la figura 3.1. De aquí en adelante resaltaremos las nuevas características clave en cada listado de código, como se muestra en las líneas 10-11 de la figura 3.14.

La aplicación es casi idéntica a la figura 3.1. Sólo veremos los cambios. La línea 2

```
// Impresión de una línea de texto mediante varias instrucciones.
```

es un comentario que indica el propósito de esta aplicación. La línea 5 comienza la declaración de la clase `Bienvenido2`.

Las líneas 10-11 del método `Main`

```
Console.Write( "¡Bienvenido a " );
Console.WriteLine( "la programación en C#!" );
```

muestran una línea de texto en la ventana de consola. La primera instrucción utiliza el método ***`Write`*** de `Console` para mostrar una cadena. A diferencia de `WriteLine`, después de mostrar su argumento, el método `Write` no posiciona el cursor de la pantalla al comienzo de la siguiente línea en la ventana de consola; el siguiente carácter que muestre la aplicación aparecerá justo después del último carácter que muestre `Write`. Por ende, la línea 11 posiciona el primer carácter en su argumento (la letra "`C`") justo después del último carácter que muestra la línea 10 (el carácter de espacio que va antes del carácter de doble comilla de cierre de la cadena). Cada instrucción `Write` continúa mostrando caracteres desde donde la última instrucción `Write` mostró su último carácter.

Mostrar varias líneas de texto mediante una sola instrucción

Una sola instrucción puede mostrar varias líneas mediante el uso de caracteres de nueva línea, que indican a los métodos `Write` y `WriteLine` de `Console` cuándo deben posicionar el cursor de la pantalla en el comienzo de la siguiente línea en la ventana de consola. Al igual que los caracteres de espacio y los tabuladores, los caracteres de nueva línea son caracteres de espacio en blanco. La aplicación de la figura 3.15 muestra cuatro líneas de texto mediante el uso de caracteres de nueva línea, para indicar cuándo se debe comenzar cada nueva línea.

```
1  // Fig. 3.14: Bienvenido2.cs
2  // Impresión de una línea de texto mediante varias instrucciones.
3  using System;
4
5  public class Bienvenido2
6  {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10       Console.Write( "¡Bienvenido a " );
11       Console.WriteLine( "la programación en C#!" );
12    } // fin del método Main
13 } // fin de la clase Bienvenido2
```

```
¡Bienvenido a la programación en C#!
```

Figura 3.14 | Impresión de una línea de texto mediante varias instrucciones.

```
1  // Fig. 3.15: Bienvenido3.cs
2  // Impresión de varias líneas mediante una sola instrucción.
3  using System;
4
5  public class Bienvenido3
6  {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10       Console.WriteLine( "¡Bienvenido\na la\nprogramación en\nC#!" );
11    } // fin del método Main
12 } // fin de la clase Bienvenido3
```

```
¡Bienvenido
a la
programación en
C#!
```

Figura 3.15 | Impresión de varias líneas mediante una sola instrucción.

La mayor parte de la aplicación es idéntica a las aplicaciones de las figuras 3.1 y 3.14, por lo que aquí sólo hablaremos de los cambios. La línea 2

```
// Impresión de varias líneas mediante una sola instrucción.
```

es un comentario que indica el propósito de esta aplicación. La línea 5 comienza la declaración de la clase `Bienvenido3`.

La línea 10

```
Console.WriteLine( "¡Bienvenido\na la\nprogramación en\nC#!" );
```

muestra cuatro líneas de texto separadas en la ventana de consola. Por lo general, los caracteres en una cadena se muestran en forma exacta a como aparecen en las comillas dobles. Sin embargo, los dos caracteres \ y n (que se repiten tres veces en la instrucción) no aparecen en la pantalla. A la ***barra diagonal inversa*** (\) se le conoce como ***carácter de escape*** y sirve para indicar a C# que hay un "carácter especial" en la cadena. Cuando aparece una barra diagonal inversa en una cadena de caracteres, C# combina el siguiente carácter con la barra diagonal inversa para formar una ***secuencia de escape***. La secuencia de escape `\n` representa el ***carácter de nueva línea***. Cuando aparece un carácter de nueva línea en una cadena que se va a imprimir en pantalla mediante métodos de la clase `Console`, este carácter hace que el cursor de la pantalla se desplace hasta el comienzo de la siguiente línea en la ventana de consola. La figura 3.16 lista varias secuencias de escape comunes y describe cómo afectan a la visualización de caracteres en la ventana de consola.

3.5 Formato del texto con `Console.Write` y `Console.WriteLine`

Los métodos `Write` y `WriteLine` de `Console` también tienen la capacidad de mostrar datos con formato. La figura 3.17 imprime en pantalla las cadenas `"¡Bienvenido a"` y `"la programación en C#!"` mediante el uso de `WriteLine`.

La línea 10

```
Console.WriteLine( "{0}\n{1}", "¡Bienvenido a", "la programación en C#!" );
```

llama al método `Console.WriteLine` para mostrar la salida de la aplicación. La llamada al método especifica tres argumentos. Cuando un método requiere varios argumentos, éstos van separados por ***comas*** (,); a lo que se le conoce como ***lista separada por comas***.

Secuencia de escape	Descripción
`\n`	Nueva línea. Posiciona el cursor de la pantalla en el comienzo de la siguiente línea.
`\t`	Tabulación horizontal. Desplaza el cursor de la pantalla a la siguiente marca de tabulación.
`\r`	Retorno de carro. Posiciona el cursor de la pantalla en el comienzo de la línea actual; no avanza el cursor a la siguiente línea. Cualquier carácter que se imprima en pantalla después del retorno de carro sobrescribirá a los caracteres que se hayan impreso antes en esa línea.
`\\`	Barra diagonal inversa. Se utiliza para colocar un carácter de barra diagonal inversa en una cadena.

`\"`	Doble comilla. Se utiliza para colocar un carácter de doble comilla (") en una cadena. Por ejemplo,
	`Console.Write( "\"entre comillas\"" );`
	se muestra en pantalla como
	`"entre comillas"`

Figura 3.16 | Algunas secuencias de escape comunes.

Buena práctica de programación 3.7

Coloque un espacio después de cada coma (,) en una lista de argumentos para que las aplicaciones tengan mejor legibilidad.

Recuerde que todas las instrucciones terminan con un punto y coma (;). Por lo tanto, la línea 10 sólo representa una instrucción. Las instrucciones largas pueden dividirse en muchas líneas, pero hay algunas restricciones.

Error común de programación 3.5

Si se divide una instrucción a la mitad de un identificador o una cadena, se produce un error de sintaxis.

El primer argumento del método `WriteLine` es una ***cadena de formato***, la cual puede consistir de ***texto fijo*** y ***elementos de formato***. El método `WriteLine` imprime en pantalla el texto fijo como se demostró en la figura 3.1. Cada elemento de formato es un receptáculo para un valor. Los elementos de formato también pueden incluir información de formato opcional.

Los elementos de formato van encerrados entre llaves y contienen una secuencia de caracteres, que indican al método qué argumento debe utilizar y cómo darle formato. Por ejemplo, el elemento de formato `{0}` es un

```
1  // Fig. 3.17: Bienvenido4.cs
2  // Impresión de varias líneas de texto con formato de cadenas.
3  using System;
4
5  public class Bienvenido4
6  {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10       Console.WriteLine( "{0}\n{1}", "¡Bienvenido a", "la programación en C#!" );
11    } // fin del método Main
12 } // fin de la clase Bienvenido4
```

```
¡Bienvenido a
la programación en C#!
```

Figura 3.17 | Impresión de varias líneas de texto con formato de cadenas.

receptáculo para el primer argumento adicional (ya que C# empieza a contar desde 0), `{1}` es un receptáculo para el segundo argumento, etc. La cadena de formato en la línea 10 especifica que `WriteLine` debe imprimir en pantalla dos argumentos y que el primero debe ir seguido de un carácter de nueva línea. Por lo tanto, este ejemplo sustituye la cadena `"¡Bienvenido a"` por el `{0}` y la cadena `"la programación en C#!"` por el `{1}`. En la salida en pantalla se muestran dos líneas de texto. Como por lo general las llaves en una cadena con formato indican un receptáculo para la sustitución de texto, debe escribir dos llaves izquierdas (`{{`) o dos llaves derechas (`}}`) para poder insertar una llave izquierda o derecha respectivamente en una cadena con formato. Conforme sea necesario, iremos presentando características adicionales de formato en nuestros ejemplos.

3.6 Otra aplicación en C#: suma de enteros

Nuestra siguiente aplicación lee (o recibe como entrada) dos ***enteros*** (números como −22, 7, 0 y 1024) que escribe un usuario en el teclado, calcula la suma de los valores y muestra el resultado. Esta aplicación debe llevar el registro de los números que suministra el usuario para realizar el cálculo más adelante. Las aplicaciones recuerdan números y otros datos en la memoria de la computadora y acceden a ellos a través de ciertos elementos, conocidos como ***variables***. La aplicación de la figura 3.18 muestra estos conceptos. En la salida de ejemplo, resaltamos, en negrita, los datos que escribe el usuario desde el teclado.

Las líneas 1-2

```
// Fig. 3.18: Suma.cs
// Muestra la suma de dos números que se introducen desde el teclado.
```

indican el número de la figura, el nombre del archivo y el propósito de la aplicación.

La línea 5

```
public class Suma
```

comienza la declaración de la clase `Suma`. Recuerde que el cuerpo de cada declaración de clase comienza con una llave izquierda de apertura (línea 6) y termina con una llave derecha de cierre (línea 26).

La aplicación comienza su ejecución con el método `Main` (líneas 8-25). La llave izquierda (línea 9) marca el comienzo del cuerpo de `Main` y la correspondiente llave derecha (línea 25) marca el final. Observe que el método `Main` tiene un nivel de sangría dentro del cuerpo de la clase `Suma` y que el código en el cuerpo de `Main` tiene otro nivel de sangría, para mejorar la legibilidad.

```
1  // Fig. 3.18: Suma.cs
2  // Muestra la suma de dos números que se introducen desde el teclado.
3  using System;
4
5  public class Suma
6  {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10       int numero1; // declara el primer número a sumar
11       int numero2; // declara el segundo número a sumar
12       int suma; // declara la suma de numero1 y numero2
13
14       Console.Write( "Escriba el primer entero: " ); // mensaje para el usuario
15       // lee el primer número del usuario
16       numero1 = Convert.ToInt32( Console.ReadLine() );
17
18       Console.Write( "Escriba el segundo entero: " ); // mensaje para el usuario
19       // lee el segundo número del usuario
```

Figura 3.18 | Impresión en pantalla de la suma de dos números introducidos desde el teclado. (Parte 1 de 2).

```
20          numero2 = Convert.ToInt32( Console.ReadLine() );
21
22          suma = numero1 + numero2; // suma los números
23
24          Console.WriteLine( "La suma es {0}", suma ); // muestra la suma
25       } // fin del método Main
26    } // fin de la clase Suma
```

```
Escriba el primer entero: 45
Escriba el segundo entero: 72
La suma es 117
```

Figura 3.18 | Impresión en pantalla de la suma de dos números introducidos desde el teclado. (Parte 2 de 2).

La línea 10

```
int numero1; // declara el primer número a sumar
```

es una ***instrucción de declaración de variable*** (también llamada ***declaración***), la cual especifica el nombre y el tipo de una variable (`numero1`) que se utilizará en esta aplicación. El nombre de una variable permite que la aplicación acceda al valor de esa variable en memoria; el nombre puede ser cualquier identificador válido (en la sección 3.2 podrá consultar los requerimientos de nomenclatura para los identificadores). El tipo de una variable especifica qué tipo de información se almacena en esa ubicación en memoria. Al igual que las demás instrucciones, las de declaración terminan con un punto y coma (`;`).

La declaración en la línea 10 especifica que la variable llamada `numero1` es de tipo ***`int`***: puede almacenar valores ***enteros*** (números como `7`, `-11`, `0` y `31914`). El rango de valores para un `int` es de –2,147,483,648 (`int.MinValue`) a +2,147,483,647 (`int.MaxValue`). Pronto hablaremos sobre los tipos ***`float`***, ***`double`*** y ***`decimal`*** para especificar números reales, y del tipo ***`char`*** para especificar caracteres. Los números reales contienen puntos decimales, como `3.4`, `0.0` y `-11.19`. Las variables de tipo `float` y `double` almacenan, en memoria, aproximaciones de números reales. Las variables de tipo `decimal` almacenan números reales con precisión (hasta 28-29 dígitos significativos), por lo que se utilizan con frecuencia en cálculos monetarios. Las variables de tipo `char` representan caracteres individuales, como una letra mayúscula (por ejemplo, `A`), un dígito (por ejemplo, `7`), un carácter especial (por ejemplo, `*` o `%`) o una secuencia de escape (por ejemplo, el carácter de nueva línea, `\n`). Por lo general, a los tipos como `int`, `float`, `double`, `decimal` y `char` se les conoce como ***tipos simples***. Los nombres de tipo simple son palabras clave y deben aparecer todos en minúscula. En el apéndice I se sintetizan las características de los trece tipos simples (`bool`, `byte`, `sbyte`, `char`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double` y `decimal`).

Las instrucciones de declaración de variables en las líneas 11-12

```
int numero2; // declara el segundo número a sumar
int suma; // declara la suma de numero1 y numero2
```

declaran de manera similar las variables `numero2` y `suma` como de tipo `int`.

Las instrucciones de declaración de variables pueden dividirse en varias líneas, con los nombres de las variables separados por comas (es decir, una lista separada por comas de nombres de variables). Pueden declararse distintas variables del mismo tipo en una o en varias declaraciones. Por ejemplo, las líneas 10-12 también pueden escribirse de la siguiente manera:

```
int numero1, // declara el primer número a sumar
    numero2, // declara el segundo número a sumar
    suma; // declara la suma de numero1 y numero2
```

Buena práctica de programación 3.8

Declare cada variable en una línea separada. Este formato permite insertar fácilmente un comentario a continuación de cada declaración.

Buena práctica de programación 3.9

*Seleccionar nombres de variables significativos ayuda a que un programa se **autodocumente** (es decir, que sea más fácil entender el código con sólo leerlo, en lugar de leer manuales o ver un número excesivo de comentarios).*

Buena práctica de programación 3.10

*Por convención, los identificadores de nombres de variables empiezan con una letra minúscula y cada una de las palabras en el nombre, que van después de la primera, deben empezar con una letra mayúscula. A esta convención se le conoce como **estilo de mayúsculas/minúsculas Camel**.*

La línea 14

```
Console.Write( "Escriba el primer entero: " ); // mensaje para el usuario
```

utiliza `Console.Write` para mostrar el mensaje `"Escriba el primer entero: "`. Este mensaje se llama ***indicador***, ya que indica al usuario que debe realizar una acción específica.

La línea 16

```
numero1 = Convert.ToInt32( Console.ReadLine() );
```

funciona en dos pasos. Primero llama al método ***`ReadLine`*** de `Console`. Este método espera a que el usuario escriba una cadena de caracteres en el teclado y que oprima *Intro* para enviar la cadena a la aplicación. Después, la cadena se utiliza como un argumento para el método ***`ToInt32`*** de la clase ***`Convert`***, el cual convierte esta secuencia de caracteres en datos de tipo `int`. Como dijimos antes en este capítulo, algunos métodos realizan una tarea y después devuelven el resultado. En este caso, el método `ToInt32` devuelve la representación `int` de la entrada del usuario.

Técnicamente, el usuario puede escribir cualquier cosa como valor de entrada. `ReadLine` lo aceptará y lo pasará al método `ToInt32`. Este método asume que la cadena contiene un valor entero válido. En esta aplicación, si el usuario escribe un valor no entero, se producirá un error lógico en tiempo de ejecución y la aplicación terminará. En el capítulo 12, Manejo de excepciones, veremos cómo hacer que sus aplicaciones sean más robustas al permitir que manejen dichos errores y continúen ejecutándose. A esto también se le conoce como hacer que su aplicación sea ***tolerante a fallas***.

En la línea 16, el resultado de la llamada al método `ToInt32` (un valor `int`) se coloca en la variable `numero1` mediante el uso del ***operador de asignación***, `=`. La instrucción se lee como "`numero1` obtiene el valor devuelto por `Convert.ToInt32`". Al operador `=` se le denomina ***operador binario***, ya que tiene dos ***operandos***: `numero1` y el resultado de la llamada al método `Convert.ToInt32`. Esta instrucción se llama ***instrucción de asignación***, ya que asigna un valor a una variable. Todo lo que esté a la derecha del operador de asignación (`=`) siempre se evalúa antes de que se realice la asignación.

Buena práctica de programación 3.11

Coloque espacios en ambos lados de un operador binario para hacerlo que resalte y que el código sea más legible.

La línea 18

```
Console.Write( "Escriba el segundo entero: " ); // mensaje para el usuario
```

pide al usuario que escriba el segundo entero. La línea 20

```
numero2 = Convert.ToInt32( Console.ReadLine() );
```

lee un segundo entero y lo asigna a la variable `numero2`.

La línea 22

```
suma = numero1 + numero2; // suma los números
```

es una instrucción de asignación que calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable suma mediante el uso del operador de asignación, `=`. La mayoría de los cálculos se realiza en instrucciones

de asignación. Cuando la aplicación encuentra el operador de suma, utiliza los valores almacenados en las variables `numero1` y `numero2` para realizar el cálculo. En la instrucción anterior, el operador de suma es un operador binario; sus dos ***operandos*** son `numero1` y `numero2`. A las porciones de las instrucciones que contienen cálculos se les llama ***expresiones***. De hecho, una expresión es cualquier porción de una instrucción que tiene un valor asociado. Por ejemplo, el valor de la expresión `numero1 + numero2` es la suma de los números. De manera similar, el valor de la expresión `Console.ReadLine()` es la cadena de caracteres que escribe el usuario.

Después de realizar el cálculo, la línea 24

```
Console.WriteLine( "La suma es {0}", suma ); // muestra la suma
```

utiliza el método `Console.WriteLine` para mostrar la `suma`. El elemento de formato `{0}` es un receptáculo para el primer argumento después de la cadena de formato. Aparte del elemento de formato `{0}`, el resto de los caracteres en la cadena de formato son texto fijo. Por lo tanto, el método `WriteLine` muestra `"La suma es"`, seguida del valor de suma (en la posición del elemento de formato `{0}`) y una nueva línea.

Los cálculos también pueden realizarse dentro de instrucciones de salida. Podríamos haber combinado las instrucciones en las líneas 22 y 24 en la instrucción

```
Console.WriteLine( "La suma es {0}", ( numero1 + numero2 ) );
```

Los paréntesis alrededor de la expresión `numero1 + numero2` no son requeridos; se incluyen para enfatizar que el valor de la expresión `numero1 + numero2` se imprime en pantalla en la posición del elemento de formato `{0}`.

3.7 Conceptos sobre memoria

Los nombres de las variables como `numero1`, `numero2` y `suma` en realidad corresponden a ***ubicaciones*** en la memoria de la computadora. Cada variable tiene un ***nombre***, un ***tipo***, un ***tamaño*** y un ***valor***.

En la aplicación de suma de la figura 3.18, cuando la instrucción (línea 16)

```
numero1 = Convert.ToInt32( Console.ReadLine() );
```

se ejecuta, el número escrito por el usuario se coloca en una ***ubicación de memoria*** a la que el compilador asigna el nombre `numero1`. Suponga que el usuario escribe **45**. La computadora coloca ese valor entero en la ubicación `numero1`, como se muestra en la figura 3.19. Siempre que se coloca un valor en una ubicación de memoria, ese valor sustituye al anterior en esa ubicación, y el valor anterior se pierde.

Cuando la instrucción (línea 20)

```
numero2 = Convert.ToInt32( Console.ReadLine() );
```

se ejecuta, suponga que el usuario escribe `72`. La computadora coloca ese valor entero en la ubicación `numero2`. La memoria ahora aparece como se muestra en la figura 3.20.

Después de que la aplicación de la figura 3.18 obtiene valores para `numero1` y `numero2`, los suma y coloca el resultado en la variable suma. La instrucción (línea 22)

```
suma = numero1 + numero2; // suma los números
```

realiza la suma y después sustituye el valor anterior de suma. Después de calcular el valor de suma, la memoria aparece como se muestra en la figura 3.21. Observe que los valores de `numero1` y `numero2` aparecen en la misma forma como se utilizaron en el cálculo de suma. Estos valores se utilizaron, pero no se destruyeron, cuando la computadora realizó el cálculo; cuando se lee un valor de una ubicación de memoria, el proceso es no destructivo.

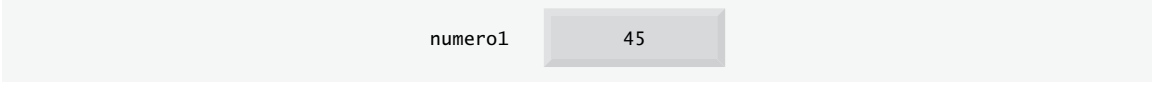

Figura 3.19 | Ubicación de memoria que muestra el nombre y el valor de la variable `numero1`.

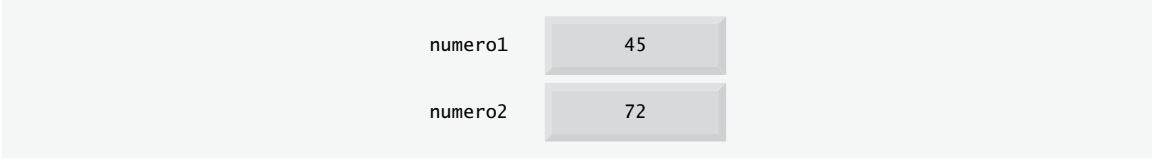

Figura 3.20 | Las ubicaciones de memoria después de almacenar valores para `numero1` y `numero2`.

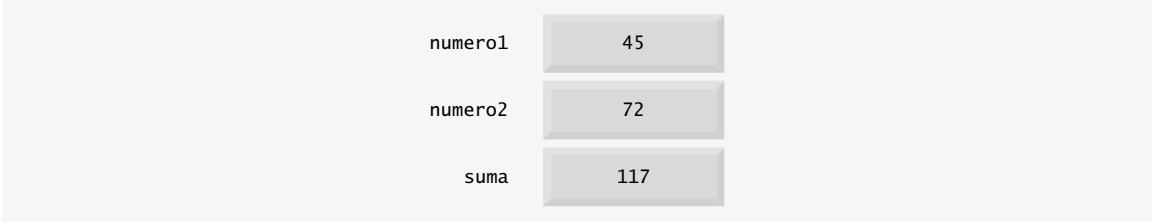

Figura 3.21 | Las ubicaciones de memoria, después de calcular y almacenar la suma de `numero1` y `numero2`.

3.8 Aritmética

La mayoría de las aplicaciones realiza cálculos aritméticos. Los ***operadores aritméticos*** se sintetizan en la figura 3.22. Observe el uso de varios símbolos especiales que no se utilizan en algebra. El ***asterisco*** (`*`) indica la multiplicación y el ***signo porcentual*** (`%`) es el ***operador residuo*** (en algunos lenguajes se le llama módulo), los cuales veremos en breve. Los operadores aritméticos de la figura 3.22 son operadores binarios; por ejemplo, la expresión `f + 7` contiene el operador binario `+` y los dos operandos `f` y `7`.

La ***división de enteros*** produce un cociente entero; por ejemplo, la expresión `7 / 4` se evalúa como `1` y la expresión `17 / 5` se evalúa como `3`. Cualquier parte fraccional en una división de enteros se descarta (es decir, se trunca); no se realiza ningún redondeo. C# cuenta con el operador residuo (`%`), el cual produce el residuo después de la división. La expresión `x % y` produce el residuo después de que `x` se divide entre `y`. Por lo tanto, `7 % 4` produce `3` y `17 % 5` produce `2`. Por lo general este operador se utiliza con operandos enteros, pero también puede utilizarse con operandos tipo `float`, `double` y `decimal`. En capítulos posteriores consideraremos varias aplicaciones interesantes del operador residuo, como determinar si un número es múltiplo de otro.

Las expresiones aritméticas deben escribirse en ***formato de línea recta*** para facilitar la introducción de aplicaciones en la computadora. Por ende, las expresiones como "`a` dividida entre `b`" deben escribirse como `a / b`, de manera que todas las constantes, variables y operadores aparezcan en una línea recta. Por lo general, los compiladores no aceptan la siguiente notación algebraica:

$$\frac{a}{b}$$

Operación de C#	Operador aritmético	Expresión algebraica	Expresión en C#
Suma	`+`	$f + 7$	`f + 7`
Resta	`-`	$p - c$	`p - c`
Multiplicación	`*`	$b \cdot m$	`b * m`
División	`/`	x/y o $\frac{x}{y}$ o $x \div y$	`x / y`
Residuo	`%`	$R \bmod s$	`r % s`

Figura 3.22 | Operadores aritméticos.

Los paréntesis se utilizan para agrupar términos en expresiones de C#, de la misma forma que en las expresiones algebraicas. Por ejemplo, para multiplicar `a` por la cantidad `b + c`, escribimos

```
a * ( b + c )
```

Si una expresión contiene ***paréntesis anidados***, como

```
( ( a + b ) * c )
```

la expresión en el conjunto más interno de paréntesis (`a + b` en este caso) se evalúa primero.

C# aplica los operadores en las expresiones aritméticas en una secuencia precisa, la cual se determina en base a las siguientes ***reglas de precedencia de operadores***, que por lo general son las mismas que las que se siguen en álgebra (figura 3.23):

1. Primero se aplican las operaciones de multiplicación, división y residuo. Si una expresión contiene varias de estas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.
2. Después se aplican las operaciones de suma y resta. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de suma y restas tienen el mismo nivel de precedencia.

Estas reglas permiten a C# aplicar operadores en el orden correcto. Cuando decimos que los operadores se aplican de izquierda a derecha, nos referimos a su ***asociatividad***. Más adelante verá que algunos operadores se asocian de derecha a izquierda. La figura 3.23 sintetiza estas reglas de precedencia de operadores. Expandiremos la tabla a medida que introduzcamos más operadores. En el apéndice A se incluye una tabla completa de precedencia.

Ahora consideremos varias expresiones en vista de las reglas de precedencia de los operadores. Cada ejemplo lista una expresión algebraica y su equivalente en C#. El siguiente es un ejemplo de una media aritmética (promedio) de cinco términos:

Álgebra: $m = \frac{a + b + c + d + e}{5}$

C#:

```
m = ( a + b + c + d + e ) / 5;
```

Los paréntesis son requeridos, ya que la división tiene mayor precedencia que la suma. Toda la cantidad `( a + b + c + d + e )` debe dividirse entre `5`. Si se omiten los paréntesis por error, obtenemos `a + b + c + d + e / 5`, lo cual se evalúa como

$$a + b + c + d + \frac{e}{5}$$

Operador(es)	Operación(es)	Orden de evaluación (asociatividad)
Se evalúa primero		
*	Multiplicación	Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
/	División	
%	Residuo	
Se evalúa después		
+	Suma	Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
-	Resta	

Figura 3.23 | Precedencia de los operadores aritméticos.

A continuación se muestra un ejemplo de la ecuación de una línea recta:

Álgebra: $y = mx + b$

C#:
```
y = m * x + b;
```

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene mayor precedencia que la suma. La asignación se realiza al último, ya que tiene una menor precedencia que la multiplicación o la suma.

El siguiente ejemplo contiene operaciones de residuo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr\%q + w/x - y$

C#:
```
z = p * r % q + w / x - y;
```

6 1 2 4 3 5

Los números dentro de los círculos bajo la instrucción indican el orden en el que C# aplica los operadores. Las operaciones de multiplicación, residuo y división se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma y la resta. Las operaciones de suma y resta se evalúan a continuación. Estas operaciones también se aplican de izquierda a derecha.

Para desarrollar una mejor comprensión de las reglas de precedencia de los operadores, considere la evaluación de un polinomio de segundo grado ($y = ax^2 + bx + c$):

```
y = a * x * x + b * x + c;
```

6 1 2 4 3 5

Los números dentro de los círculos indican el orden en el que C# aplica los operadores. Las operaciones de multiplicación se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma. Las operaciones de suma se evalúan a continuación y se aplican de izquierda a derecha. Como no hay operador aritmético para la exponenciación en C#, x^2 se representa como `x * x`. La sección 6.4 muestra una alternativa para realizar la exponenciación en C#.

Suponga que `a`, `b`, `c` y `x` en el polinomio anterior de segundo grado se inicializan (reciben valores) de la siguiente manera: `a = 2`, `b = 3`, `c = 7` y `x = 5`. La figura 3.24 ilustra el orden en el que se aplican los operadores.

Al igual que en álgebra, es aceptable colocar paréntesis innecesarios en una expresión para mejorar su legibilidad. A éstos se les llama ***paréntesis redundantes***. Por ejemplo, podrían utilizarse paréntesis en la instrucción de asignación anterior para resaltar sus términos de la siguiente manera:

```
y = ( a * x * x ) + ( b * x ) + c;
```

3.9 Toma de decisiones: operadores de igualdad y relacionales

Una ***condición*** es una expresión que puede ser ***verdadera*** o ***falsa***. En esta sección introducimos una versión simple de la ***instrucción `if`*** de C#, que permite que una aplicación tome una ***decisión*** con base en el valor de una condición. Por ejemplo, la condición "calificación es mayor o igual a 60" determina si un estudiante pasó una prueba. Si la condición en una instrucción `if` es verdadera, se ejecuta el cuerpo de la instrucción `if`. Si la condición es falsa, el cuerpo no se ejecuta. En breve veremos un ejemplo.

Las condiciones en las instrucciones `if` pueden formarse mediante el uso de los ***operadores de igualdad*** (`==` y `!=`) y los ***operadores relacionales*** (`>`, `<`, `>=` y `<=`), los cuales se sintetizan en la figura 3.25. Los dos operadores de igualdad (== y !=) tienen cada uno el mismo nivel de precedencia, los operadores relacionales (>, <, >= y <=) tienen cada uno el mismo nivel de precedencia y los operadores de igualdad tienen menor precedencia que los operadores relacionales. Todos se asocian de izquierda a derecha.

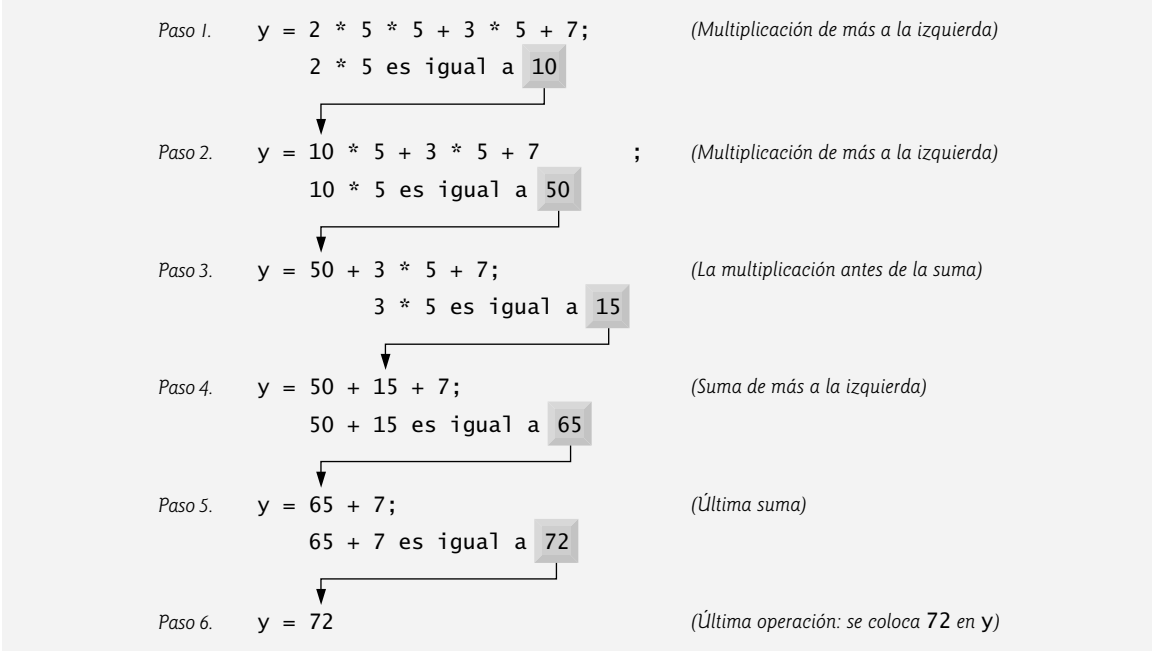

Figura 3.24 | Orden en el que se evalúa un polinomio de segundo grado.

La aplicación de la figura 3.26 utiliza seis instrucciones `if` para comparar dos enteros introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es verdadera, se ejecuta la instrucción de asignación asociada con esa instrucción `if`. La aplicación utiliza la clase `Console` para solicitar y leer dos líneas de texto del usuario, extrae los enteros de ese texto mediante el método `ToInt32` de la clase `Convert` y los almacena en las variables `numero1` y `numero2`. Después la aplicación compara los números y muestra los resultados de las comparaciones que son verdaderas.

La declaración de la clase `Comparacion` comienza en la línea 6

```
public class Comparacion
```

El método `Main` de la clase (líneas 9-39) comienza la ejecución de la aplicación.

Operadores estándar algebraicos de igualdad y relacionales	Operador de igualdad o relacional de C#	Operador de igualdad o relacional de C#	Significado de la condición C#
Operadores de igualdad			
=	==	x == y	x es igual a y
≠	!=	x != y	x no es igual a y
Operadores relacionales			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y

Figura 3.25 | Operadores de igualdad y relacionales.

Las líneas 11-12

```
int numero1; // declara el primer número a comparar
int numero2; // declara el segundo número a comparar
```

declaran las variables `int` que se utilizan para almacenar los valores introducidos por el usuario.

Las líneas 14-16

```
// pide al usuario y lee el primer número
Console.Write( "Escriba el primer entero: " );
numero1 = Convert.ToInt32( Console.ReadLine() );
```

piden al usuario que escriba el primer entero y reciben ese valor como entrada. El valor de entrada se almacena en la variable `numero1`.

```
1  // Fig. 3.26: Comparacion.cs
2  // Comparación de enteros mediante el uso de instrucciones if, operadores de
3  // igualdad y operadores relacionales.
4  using System;
5
6  public class Comparacion
7  {
8     // El método Main comienza la ejecución de la aplicación de C#
9     public static void Main( string[] args )
10    {
11       int numero1; // declara el primer número a comparar
12       int numero2; // declara el segundo número a comparar
13
14       //pide al usuario y lee el primer número
15       Console.Write( "Escriba el primer entero: " );
16       numero1 = Convert.ToInt32( Console.ReadLine() );
17
18       //pide al usuario y lee el segundo número
19       Console.Write( "Escriba el segundo entero: " );
20       numero2 = Convert.ToInt32( Console.ReadLine() );
21
22       if ( numero1 == numero2 )
23          Console.WriteLine( "{0} == {1}", numero1, numero2 );
24
25       if ( numero1 != numero2 )
26          Console.WriteLine( "{0} != {1}", numero1, numero2 );
27
28       if ( numero1 < numero2 )
29          Console.WriteLine( "{0} < {1}", numero1, numero2 );
30
31       if ( numero1 > numero2 )
32          Console.WriteLine( "{0} > {1}", numero1, numero2 );
33
34       if ( numero1 <= numero2 )
35          Console.WriteLine( "{0} <= {1}", numero1, numero2 );
36
37       if ( numero1 >= numero2 )
38          Console.WriteLine( "{0} >= {1}", numero1, numero2 );
39    } // fin del método Main
40 } // fin de la clase Comparacion
```

Figura 3.26 | Comparación de enteros mediante el uso de instrucciones `if`, operadores de igualdad y operadores relacionales. (Parte 1 de 2).

```
Escriba el primer entero: 42
Escriba el segundo entero: 42
42 == 42
42 <= 42
42 >= 42
```

```
Escriba el primer entero: 1000
Escriba el segundo entero: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Escriba el primer entero: 2000
Escriba el segundo entero: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Figura 3.26 | Comparación de enteros mediante el uso de instrucciones `if`, operadores de igualdad y operadores relacionales. (Parte 2 de 2).

Las líneas 18-20

```
// pide al usuario y lee el segundo número
Console.Write( "Escriba el segundo entero: " );
numero2 = Convert.ToInt32( Console.ReadLine() );
```

realizan la misma tarea, sólo que el valor de entrada se almacena en la variable `numero2`.

Las líneas 22-23

```
if ( numero1 == numero2 )
   Console.WriteLine( "{0} == {1}", numero1, numero2 );
```

comparan los valores de las variables `numero1` y `numero2` para determinar si son iguales. Una instrucción `if` siempre comienza con la palabra clave `if`, seguida de una condición entre paréntesis. Una instrucción `if` espera una instrucción en su cuerpo. La sangría de la instrucción del cuerpo que se muestra aquí no es requerida, pero aumenta la legibilidad del código al enfatizar que la instrucción de la línea 23 es parte de la instrucción `if` que comienza en la línea 22. La línea 23 se ejecuta sólo si los números almacenados en las variables `numero1` y `numero2` son iguales (es decir, que la condición sea verdadera). Las instrucciones `if` en las líneas 25-26, 28-29, 31-32, 34-35 y 37-38 comparan a `numero1` y `numero2` mediante los operadores `!=`, `<`, `>`, `<=` y `>=`, en forma respectiva. Si la condición en cualquiera de las instrucciones `if` es verdadera, se ejecuta la correspondiente instrucción del cuerpo.

Error común de programación 3.6

Olvidar los paréntesis izquierdo y/o derecho para la condición en una instrucción `if` es un error de sintaxis; los paréntesis son requeridos.

Error común de programación 3.7

Confundir el operador de igualdad, `==`, con el de asignación, `=`, puede provocar un error lógico o un error de sintaxis. El operador de igualdad debe leerse como "es igual a", y el de asignación debe leerse como "obtiene" u "obtiene el valor de". Para evitar confusión, algunas personas leen el operador de igualdad como "doble igual a" o "igual a igual a".

Error común de programación 3.8

Si los operadores ==, !=, >= y <= contienen espacios entre sus símbolos, como en = =, ! =, > = y < =, en forma respectiva, se produce un error de sintaxis.

Error común de programación 3.9

Si se invierten los operadores !=, >= y <=, como en =!, => y =<, se produce un error de sintaxis.

Buena práctica de programación 3.12

Aplique sangría al cuerpo de una instrucción `if` *para hacer que resalte y mejorar la legibilidad de la aplicación.*

Observe que no hay punto y coma (`;`) al final de la primera línea de cada instrucción `if`. Dicho punto y coma produciría un error lógico en tiempo de ejecución. Por ejemplo,

```
if ( numero1 == numero2 );  // error lógico
   Console.WriteLine( "{0} == {1}", numero1, numero2 );
```

C# lo interpretaría en realidad como

```
if ( numero1 == numero2 )
   ;  // instrucción vacía
Console.WriteLine( "{0} == {1}", numero1, numero2 );
```

en donde el punto y coma en la línea por sí solo (a lo cual se le llama ***instrucción vacía***) es la instrucción que se ejecutará si la condición en la instrucción `if` es verdadera. Cuando se ejecuta la instrucción vacía, no se realiza ninguna tarea en la aplicación. Así, la aplicación continúa con la instrucción de salida, que siempre se ejecuta sin importar que la condición sea verdadera o falsa, ya que no forma parte de la instrucción `if`.

Error común de programación 3.10

Colocar un punto y coma inmediatamente después del paréntesis derecho de la condición en una instrucción `if` *es por lo general un error lógico.*

Observe el uso del espacio en blanco en la figura 3.26. Recuerde que por lo general el compilador ignora los caracteres en blanco, como los tabuladores, nuevas líneas y espacios. Así, las instrucciones pueden dividirse en varias líneas y es posible utilizar espacios de acuerdo a sus preferencias, sin afectar el significado de una aplicación. Es incorrecto dividir identificadores, cadenas y operadores con varios caracteres (como `>=`). En teoría, las instrucciones deben mantenerse reducidas, pero esto no siempre es posible.

Buena práctica de programación 3.13

No coloque más de una instrucción por línea en una aplicación. Este formato mejora la legibilidad.

Buena práctica de programación 3.14

Una instrucción larga puede esparcirse en varias líneas. Si una sola instrucción debe dividirse en dos o más líneas, elija puntos de división que tengan sentido, como después de una coma en una lista separada por comas, o después de un operador en una expresión larga. Si se divide una instrucción en dos o más líneas, aplique sangría a todas las líneas subsiguientes hasta el final de la instrucción.

La figura 3.27 muestra la precedencia de los operadores que presentamos en este capítulo. Los operadores se muestran de arriba hacia abajo, en orden decreciente de precedencia. Todos estos operadores, excepto el operador de asignación (=), se asocian de izquierda a derecha. La suma es asociativa a la izquierda, por lo que una expresión como `x + y + z` se evalúa como si se hubiera escrito así: `( x + y ) + z`. El operador de asignación (=) se asocia de derecha a izquierda, por lo que una expresión como `x = y = 0` se evalúa como si se hubiera escrito así: `x = (y = 0)`, lo cual, como pronto veremos, asigna primero el valor `0` a la variable `y` y después asigna el resultado de esa asignación (`0`) a `x`.

Operadores	Asociatividad	Tipo
* / %	izquierda a derecha	multiplicativa
+ -	izquierda a derecha	aditiva
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
=	derecha a izquierda	asignación

Figura 3.27 | Precedencia y asociatividad de las operaciones descritas.

Buena práctica de programación 3.15

Consulte la tabla de precedencia de operadores (en el apéndice A podrá ver la tabla completa) cuando escriba expresiones que contengan muchos operadores. Confirme que las operaciones en la expresión se realicen en el orden que espera. Si no está seguro acerca del orden de evaluación en una expresión compleja, utilice paréntesis para forzar el orden, como lo haría con las expresiones algebraicas. Tenga en cuenta que algunos operadores, como el de asignación (=), asocian de derecha a izquierda, en vez de asociar de izquierda a derecha.

3.10 (Opcional) Caso de estudio de ingeniería de software: análisis del documento de requerimientos del ATM

Ahora empezaremos nuestro caso de estudio opcional de diseño e implementación orientados a objetos. Las secciones tituladas Caso de estudio de ingeniería de software al final de éste y los siguientes capítulos lo llevarán a través de la orientación a objetos. Desarrollaremos software para un sistema simple de cajero automático (ATM), con lo cual le proporcionaremos una experiencia de diseño e implementación orientados a objetos concisa, minuciosa y completa. En los capítulos 4-9 y 11 llevaremos a cabo los diversos pasos de un proceso de diseño orientado a objetos (DOO) mediante el uso del UML, al mismo tiempo que relacionaremos estos pasos con los conceptos orientados a objetos que veremos en los capítulos. En el apéndice J implementaremos el ATM mediante el uso de las técnicas de la programación orientada a objetos (POO) en C# y presentaremos la solución completa al caso de estudio. Éste no es un ejercicio, sino una experiencia de aprendizaje de principio a fin, la cual concluiremos con un paseo detallado por el código completo de C# que implementa nuestro diseño. Con esto empezará a familiarizarse con los tipos de problemas sustanciales que se encuentran en la industria, junto con sus soluciones.

Empezaremos nuestro proceso de diseño mediante la presentación de un ***documento de requerimientos***, el cual especifica el propósito general del sistema ATM y *qué* es lo que debe hacer. A lo largo del caso de estudio nos referiremos al documento de requerimientos para determinar con precisión qué funcionalidad debe incluir el sistema.

Documento de requerimientos

Un banco local pequeño pretende instalar un nuevo cajero automático (ATM) para permitir que los usuarios (es decir, clientes del banco) realicen transacciones financieras básicas (figura 3.28). Por cuestión de simpleza, cada usuario sólo puede tener una cuenta en el banco. Los usuarios del ATM deben poder ver su saldo, retirar efectivo (es decir, sacar dinero de una cuenta) y depositar fondos (es decir, colocar dinero en una cuenta).

La interfaz de usuario del cajero automático contiene los siguientes componentes de hardware:

- una pantalla que muestra mensajes al usuario.
- un teclado numérico que recibe entrada numérica del usuario.
- un dispensador de efectivo que entrega efectivo al usuario.
- una ranura para depósitos que recibe sobres para depósitos del usuario.

El dispensador de efectivo comienza cada día cargado con 500 billetes de $20. [*Nota*: debido al alcance limitado de este caso de estudio, ciertos elementos del ATM que se describen aquí simplifican varios aspectos de uno real. Por ejemplo, con frecuencia un ATM contiene un dispositivo que lee el número de cuenta del

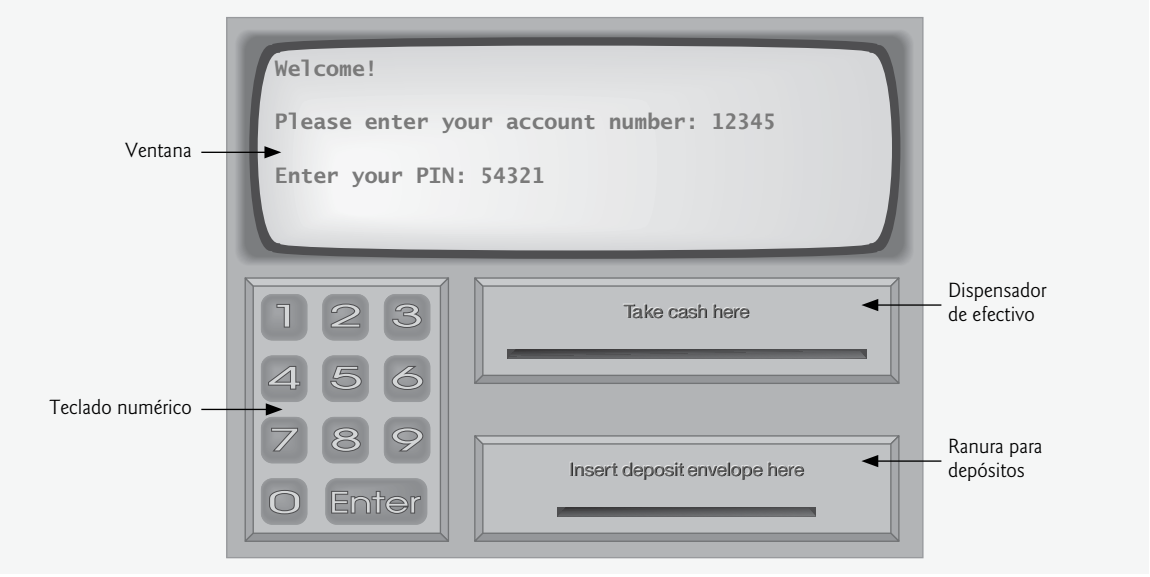

Figura 3.28 | Interfaz de usuario del cajero automático.

usuario de una tarjeta para ATM, mientras que este pide al usuario que escriba su número de cuenta en el teclado (el cual usted simulará con el teclado de su computadora personal). Además, un ATM real, por lo general, imprime un recibo de papel al final de una sesión, y toda la salida de este ATM aparece en la pantalla.]

El banco desea que usted desarrolle software para realizar las transacciones financieras que inicien los clientes del banco a través del ATM. El banco integrará posteriormente el software con el hardware del ATM. El software debe simular la funcionalidad de los dispositivos de hardware (por ejemplo: dispensador de efectivo, ranura para depósito) mediante componentes de software, pero no necesita preocuparse por cómo realizan estos dispositivos su trabajo. El hardware del ATM no se ha desarrollado aún, por lo que en vez de que usted escriba su software para que se ejecute en el ATM, deberá desarrollar una primera versión del software para que se ejecute en una computadora personal. Esta versión debe utilizar el monitor de la computadora para simular la pantalla del ATM y el teclado de la computadora para simular el teclado numérico del ATM.

Una sesión con el ATM consiste en la autenticación de un usuario (es decir, proporcionar la identidad del usuario) con base en un número de cuenta y un número de identificación personal (NIP), seguida de la creación y la ejecución de transacciones financieras. Para autenticar un usuario y realizar transacciones, el ATM debe interactuar con la base de datos de información sobre las cuentas del banco. [*Nota*: una base de datos es una colección organizada de datos almacenados en una computadora.] Para cada cuenta de banco, la base de datos almacena un número de cuenta, un NIP y un balance que indica la cantidad de dinero en la cuenta. [*Nota*: el banco planea construir sólo un ATM, por lo que no necesitamos preocuparnos porque varios ATMs accedan a la base de datos al mismo tiempo. Lo que es más, supongamos que el banco no realizará modificaciones en la información que hay en la base de datos mientras un usuario accede al ATM. Además, cualquier sistema comercial como un ATM se topa con cuestiones de seguridad con una complejidad razonable, las cuales van más allá del alcance de un curso de programación de primer o segundo semestre. No obstante, para simplificar nuestro ejemplo supongamos que el banco confía en el ATM para que acceda a la información en la base de datos y la manipule sin necesidad de medidas de seguridad considerables.]

Al acercarse al ATM, el usuario deberá experimentar la siguiente secuencia de eventos (vea la figura 3.28):

1. La pantalla muestra un mensaje de bienvenida y pide al usuario que introduzca un número de cuenta.
2. El usuario introduce un número de cuenta de cinco dígitos, mediante el uso del teclado.
3. Para fines de autenticación, la pantalla pide al usuario que introduzca su NIP (número de identificación personal) asociado con el número de cuenta especificado.

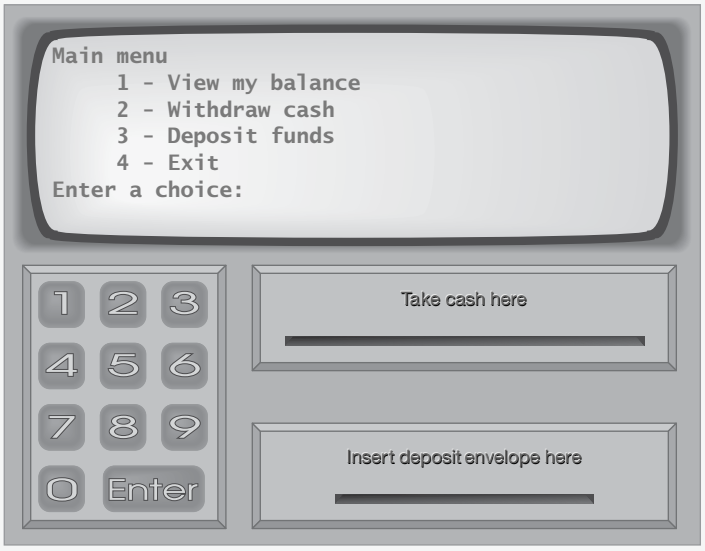

Figura 3.29 | Menú principal del ATM.

4. El usuario introduce un NIP de cinco dígitos mediante el teclado numérico.
5. Si el usuario introduce un número de cuenta válido y el NIP correcto para esa cuenta, la pantalla muestra el menú principal (figura 3.29). Si el usuario introduce un número de cuenta inválido o un NIP incorrecto, la pantalla muestra un mensaje apropiado y después el ATM regresa al *paso 1* para reiniciar el proceso de autenticación.

Una vez que el ATM autentique al usuario, el menú principal (figura 3.29) mostrará una opción numerada para cada uno de los tres tipos de transacciones: solicitud de saldo (opción 1), retiro (opción 2) y depósito (opción 3). El menú principal también mostrará una opción que permite al usuario salir del sistema (opción 4). Después, el usuario elegirá si desea realizar una transacción (oprimiendo 1, 2 o 3) o salir del sistema (4). Si el usuario introduce una opción inválida, la pantalla mostrará un mensaje de error y volverá a mostrar el menú principal.

Si el usuario oprime 1 para solicitar su saldo, la pantalla mostrará el saldo de esa cuenta bancaria. Para ello, el ATM deberá obtener el saldo de la base de datos del banco.

Las siguientes acciones se realizan cuando el usuario elige la opción 2 para hacer un retiro:

1. La pantalla muestra un menú (vea la figura 3.30) que contenga montos de retiro estándar: $20 (opción 1), $40 (opción 2), $60 (opción 3), $100 (opción 4) y $200 (opción 5). El menú también contiene la opción 6, que permite al usuario cancelar la transacción.
2. El usuario introduce la selección del menú (1-6) mediante el teclado numérico.
3. Si el monto elegido a retirar es mayor que el saldo de la cuenta del usuario, la pantalla muestra un mensaje indicando esta situación y pide al usuario que seleccione un monto más pequeño. Entonces el ATM regresa al *paso 1*. Si el monto elegido a retirar es menor o igual que el saldo de la cuenta del usuario (es decir, un monto de retiro aceptable), el ATM procede al *paso 4*. Si el usuario opta por cancelar la transacción (opción 6), el ATM muestra el menú principal (figura 3.29) y espera la entrada del usuario.
4. Si el dispensador contiene suficiente efectivo para satisfacer la solicitud, el ATM procede al *paso 5*. En caso contrario, la pantalla muestra un mensaje indicando el problema y pide al usuario que seleccione un monto de retiro más pequeño. Después el ATM regresa al *paso 1*.
5. El ATM carga (es decir, resta) el monto de retiro al saldo de la cuenta del usuario en la base de datos del banco.

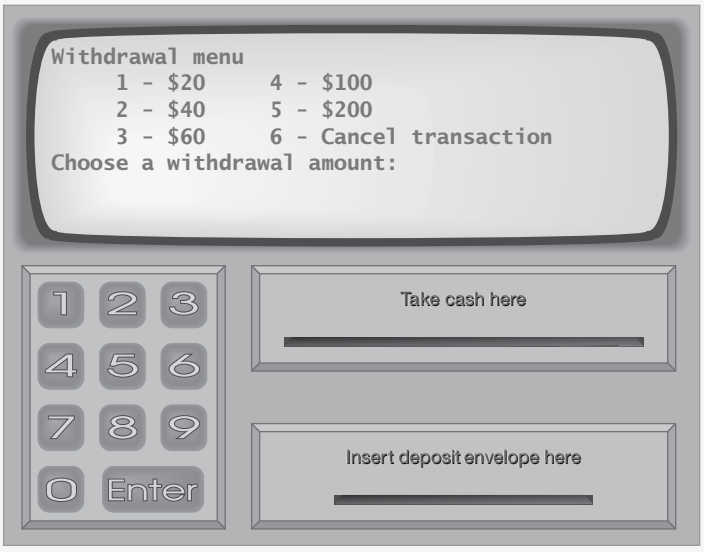

Figura 3.30 | Menú de retiro del ATM.

6. El dispensador de efectivo entrega el monto deseado de dinero al usuario.
7. La pantalla muestra un mensaje para recordar al usuario que tome el dinero.

Las siguientes acciones se realizan cuando el usuario elige la opción 3 (del menú principal) para hacer un depósito:

1. La pantalla pide al usuario que introduzca un monto de depósito o que escriba 0 (cero) para cancelar la transacción.
2. El usuario introduce un monto de depósito o 0 mediante el teclado numérico. [*Nota*: estos teclados no contienen un punto decimal o signo de moneda, por lo que el usuario no puede escribir una cantidad real (por ejemplo: $147.25), sino que debe escribir un monto de depósito en forma de número de centavos (por ejemplo: 14725). Después, el ATM divide este número entre 100 para obtener un número que represente un monto ya sea en pesos o en dólares (por ejemplo, 14725 ÷ 100 = 147.25).]
3. Si el usuario especifica un monto a depositar, el ATM procede al *paso 4*. Si elije cancelar la transacción (escribiendo 0), el ATM muestra el menú principal (figura 3.29) y espera la entrada del usuario.
4. La pantalla muestra un mensaje indicando al usuario que introduzca un sobre de depósito en la ranura para depósitos.
5. Si la ranura de depósitos recibe un sobre dentro de un plazo no mayor a 2 minutos, el ATM abona (es decir, suma) el monto de depósito al saldo de la cuenta del usuario en la base de datos del banco. [*Nota*: este dinero no está disponible de inmediato para retirarse. El banco primero debe verificar el monto de efectivo en el sobre y cualquier cheque que éste contenga debe validarse (es decir, el dinero debe transferirse de la cuenta del emisor del cheque a la cuenta del beneficiario). Cuando ocurra uno de estos eventos, el banco actualizará de manera apropiada el saldo del usuario que está almacenado en su base de datos. Esto ocurre de manera independiente al sistema ATM]. Si la ranura de depósito no recibe un sobre dentro de un plazo no mayor a dos minutos, la pantalla muestra un mensaje indicando que el sistema canceló la transacción debido a la inactividad. Después, el ATM muestra el menú principal y espera la entrada del usuario.

Una vez que el sistema ejecuta una transacción en forma exitosa, debe volver a mostrar el menú principal (figura 3.29) para que el usuario pueda realizar transacciones adicionales. Si el usuario elije salir del sistema

(opción 4), la pantalla debe mostrar un mensaje de agradecimiento y después el mensaje de bienvenida para el siguiente usuario.

Análisis del sistema de ATM

En la declaración anterior se presentó un documento de requerimientos simplificado. Por lo general, dicho documento es el resultado de un proceso detallado de ***recopilación de requerimientos***, el cual podría incluir entrevistas con usuarios potenciales del sistema y especialistas en campos relacionados con el mismo. Por ejemplo, un analista de sistemas que se contrate para preparar un documento de requerimientos para software bancario (es decir, el sistema ATM que describimos aquí) podría entrevistar a personas que hayan utilizado ATMs y a expertos financieros para obtener una mejor comprensión de *qué* es lo que debe hacer el software. El analista utilizaría la información recopilada para compilar una lista de ***requerimientos del sistema*** para guiar a los diseñadores de sistemas.

El proceso de recopilación de requerimientos es una tarea clave de la primera etapa del ciclo de vida del software. El ***ciclo de vida del software*** especifica las etapas a través de las cuales el software evoluciona desde el tiempo en que fue concebido hasta el tiempo en que se retira de su uso. Por lo general, estas etapas incluyen el análisis, diseño, implementación, prueba y depuración, despliegue, mantenimiento y retiro. Existen varios modelos de ciclo de vida del software, cada uno con sus propias preferencias y especificaciones con respecto a cuándo y qué tan a menudo deben llevar a cabo los ingenieros de software las diversas etapas. Los ***modelos de cascada*** realizan cada etapa una vez en sucesión, mientras que los ***modelos iterativos*** pueden repetir una o más etapas varias veces a lo largo del ciclo de vida de un producto.

La etapa de análisis del ciclo de vida del software se enfoca en definir con precisión el problema a resolver. Al diseñar cualquier sistema, es indudable que uno debe *resolver el problema de la manera correcta*, pero de igual manera uno debe *resolver el problema correcto*. Los analistas de sistemas recolectan los requerimientos que indican el problema específico a resolver. Nuestro documento de requerimientos describe nuestro sistema ATM simple con el suficiente detalle como para que usted no necesite pasar por una etapa de análisis exhaustiva; ya lo hicimos por usted.

Para capturar lo que debe hacer un sistema propuesto, los desarrolladores emplean a menudo una técnica conocida como ***modelado de caso-uso***. Este proceso identifica los ***casos de uso*** del sistema, cada uno de los cuales representa una capacidad distinta que el sistema provee a sus clientes. Por ejemplo, es común que los ATMs tengan varios casos de uso, como "Ver saldo de cuenta", "Retirar efectivo", "Depositar fondos", "Transferir fondos entre cuentas" y "Comprar estampas postales". El sistema ATM simplificado que construiremos en este caso de estudio requiere sólo los tres primeros casos de uso (figura 3.31).

Cada uno de los casos de uso describe un escenario común en el cual el usuario utiliza el sistema. Usted ya leyó las descripciones de los casos de uso del sistema ATM en el documento de requerimientos; las listas de pasos requeridos para realizar cada tipo de transacción (es decir, solicitud de saldo, retiro y depósito) describen los tres casos de uso de nuestro ATM: "Ver saldo de cuenta", "Retirar efectivo" y "Depositar fondos".

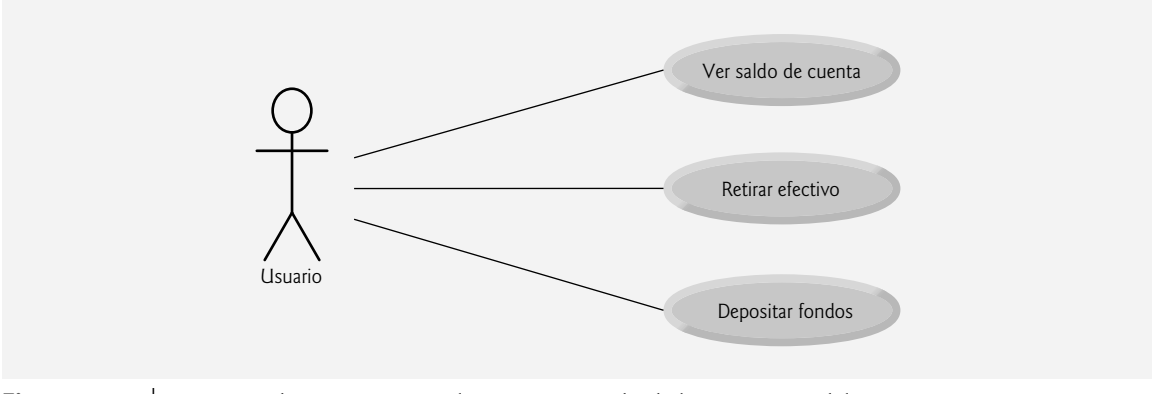

Figura 3.31 | Diagrama de caso-uso para el sistema ATM, desde la perspectiva del usuario.

Diagramas de caso-uso

Ahora presentaremos el primero de varios diagramas de UML en nuestro caso de estudio del ATM. Crearemos un ***diagrama de caso-uso*** para modelar las interacciones entre los clientes de un sistema (en este caso de estudio, los clientes del banco) y el sistema. El objetivo es mostrar los tipos de interacciones que tienen los usuarios con un sistema sin proveer los detalles; éstos se mostrarán en otros diagramas de UML (que presentaremos a lo largo del caso de estudio). A menudo, los diagramas de caso-uso se acompañan de texto informal que describe los casos de uso con más detalle; como el texto que aparece en el documento de requerimientos. Los diagramas de caso-uso se producen durante la etapa de análisis del ciclo de vida del software. En sistemas más grandes, los diagramas de caso-uso son herramientas simples pero indispensables, que ayudan a los diseñadores de sistemas a enfocarse en satisfacer las necesidades de los usuarios.

La figura 3.31 muestra el diagrama de caso-uso para nuestro sistema ATM. La figura humana representa a un ***actor***, quien define los roles que desempeña una entidad externa (como una persona u otro sistema) cuando interactúa con el sistema. Para nuestro cajero automático, el actor es un Usuario que puede ver el saldo de una cuenta, retirar efectivo y depositar fondos mediante el uso del ATM. El Usuario no es una persona real, sino que constituye los roles que pude desempeñar una persona real (al desempeñar el papel de un Usuario) mientras interactúa con el ATM. Hay que tener en cuenta que un diagrama de caso-uso puede incluir varios actores. Por ejemplo, el diagrama de caso-uso para un sistema ATM de un banco real podría incluir también un actor llamado Administrador, que rellene el dispensador de efectivo a diario.

Para identificar al actor en nuestro sistema debemos examinar el documento de requerimientos, el cual dice que "los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo y depositar fondos". El actor en cada uno de estos tres casos de uso es simplemente el Usuario que interactúa con el ATM. Una entidad externa (una persona real) desempeña el papel del Usuario para realizar transacciones financieras. La figura 3.31 muestra un actor, cuyo nombre (Usuario) aparece debajo del actor en el diagrama. UML modela cada caso de uso como un óvalo conectado a un actor con una línea sólida.

Los ingenieros de software (específicamente, los diseñadores de sistemas) deben analizar el documento de requerimientos o un conjunto de casos de uso, y diseñar el sistema antes de que los programadores lo implementen en un lenguaje de programación específico. Durante la etapa de análisis, los diseñadores de sistemas se enfocan en comprender el documento de requerimientos para producir una especificación de alto nivel que describa *qué* es lo que el sistema debe hacer. El resultado de la etapa de diseño (una ***especificación de diseño***) debe especificar *cómo* debe construirse el sistema para satisfacer estos requerimientos. En las siguientes secciones del Caso de estudio de ingeniería de software, llevaremos a cabo los pasos de un proceso simple de DOO con el sistema ATM para producir una especificación de diseño que contenga una colección de diagramas de UML y texto de apoyo. Recuerde que UML está diseñado para utilizarse con cualquier proceso de DOO. Existen muchos de esos procesos, de los cuales el más conocido es Rational Unified Process™ (RUP), desarrollado por Rational Software Corporation (ahora una división de IBM). RUP es un proceso robusto para diseñar aplicaciones a nivel industrial. Para este caso de estudio, presentaremos un proceso de diseño simplificado.

Diseño del sistema ATM

Ahora comenzaremos la etapa de diseño de nuestro sistema ATM. Un ***sistema*** es un conjunto de componentes que interactúan para resolver un problema. Por ejemplo, para realizar sus tareas designadas, nuestro sistema ATM tiene una interfaz de usuario (figura 3.28), contiene software para ejecutar transacciones financieras e interactúa con una base de datos de información de cuentas bancarias. La ***estructura del sistema*** describe los objetos del sistema y sus interrelaciones. El ***comportamiento del sistema*** describe la manera en que cambia el sistema a medida que sus objetos interactúan entre sí. Todo sistema tiene tanto estructura como comportamiento; los diseñadores deben especificar ambos. Existen diversos tipos de estructuras y comportamientos de un sistema. Por ejemplo, las interacciones entre los objetos en el sistema son distintas a las interacciones entre el usuario y el sistema, pero aun así ambas constituyen una porción del comportamiento del sistema.

UML 2 especifica 13 tipos de diagramas para documentar los modelos de un sistema. Cada tipo de diagrama modela una característica distinta de la estructura o del comportamiento de un sistema; seis tipos de diagramas se relacionan con la estructura del sistema; los siete restantes se relacionan con su comportamiento. Aquí listaremos sólo los seis tipos de diagramas que utilizaremos en nuestro caso de estudio, uno de los cuales (el diagrama de clases) modela la estructura del sistema; los otros cinco modelan el comportamiento. En el apéndice K, UML 2: Tipos de diagramas adicionales, veremos las generalidades sobre los siete tipos restantes de diagramas de UML.

1. Los ***diagramas de caso-uso***, como el de la figura 3.31, modelan las interacciones entre un sistema y sus entidades externas (actores) en términos de casos de uso (capacidades del sistema, como "Ver saldo de cuenta", "Retirar efectivo" y "Depositar fondos").
2. Los ***diagramas de clases***, que estudiará en la sección 4.11, modelan las clases o "bloques de construcción" que se utilizan en un sistema. Cada sustantivo u "objeto" que se describe en el documento de requerimientos es candidato para ser una clase en el sistema (por ejemplo, "cuenta", "teclado"). Los diagramas de clases nos ayudan a especificar las relaciones estructurales entre las partes del sistema. Por ejemplo, el diagrama de clases del sistema ATM especificará, entre otras cosas, que el ATM está compuesto físicamente de una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos.
3. Los ***diagramas de máquina de estado***, que estudiará en la sección 6.9, modelan las formas en que un objeto cambia de estado. El ***estado*** de un objeto se indica mediante los valores de todos los atributos del objeto, en un momento dado. Cuando un objeto cambia de estado, en consecuencia puede comportarse de manera distinta en el sistema. Por ejemplo, después de validar el NIP de un usuario, el ATM cambia del estado "usuario no autenticado" al estado "usuario autenticado", punto en el cual el ATM permite al usuario realizar transacciones financieras (por ejemplo, ver el saldo de su cuenta, retirar efectivo, depositar fondos).
4. Los ***diagramas de actividad***, que también estudiará en la sección 6.9, modelan la ***actividad*** de un objeto: el flujo de trabajo (secuencia de eventos) del objeto durante la ejecución del programa. Un diagrama de actividad modela las acciones que realiza el objeto y especifica el orden en el cual desempeña estas acciones. Por ejemplo, un diagrama de actividad muestra que el ATM debe obtener el saldo de la cuenta del usuario (de la base de datos de información de las cuentas del banco) antes de que la pantalla pueda mostrar el saldo al usuario.
5. Los ***diagramas de comunicación*** (llamados diagramas de colaboración en versiones anteriores de UML) modelan las interacciones entre los objetos en un sistema, con un énfasis acerca de *qué* interacciones ocurren. En la sección 8.14 aprenderá que estos diagramas muestran cuáles objetos deben interactuar para realizar una transacción en el ATM. Por ejemplo, el ATM debe comunicarse con la base de datos de información de las cuentas del banco para obtener el saldo de una cuenta.
6. Los ***diagramas de secuencia*** modelan también las interacciones entre los objetos en un sistema, pero a diferencia de los diagramas de comunicación, enfatizan *cuándo* ocurren las interacciones. En la sección 8.14 aprenderá que estos diagramas ayudan a mostrar el orden en el que ocurren las interacciones al ejecutar una transacción financiera. Por ejemplo, la pantalla pide al usuario que escriba un monto de retiro antes de dispensar el efectivo.

En la sección 4.11 seguiremos diseñando nuestro sistema ATM; ahí identificaremos las clases del documento de requerimientos. Para lograr esto, vamos a extraer sustantivos clave y frases nominales del documento de requerimientos. Mediante el uso de estas clases, desarrollaremos nuestro primer borrador del diagrama de clases que modelará la estructura de nuestro sistema ATM.

Recursos en Internet y Web

Las siguientes URLs proporcionan información sobre el diseño orientado a objetos con el UML.

`www-306.ibm.com/software/rational/uml/`
Lista preguntas frecuentes acerca del UML, proporcionado por IBM Rational.

`www.douglass.co.uk/documents/softdocwiz.com.UML.htm`
Vínculos al Diccionario del Lenguaje unificado de modelado, el cual define todos los términos utilizados en el UML.

`www.agilemodeling.com/essays/umlDiagrams.htm`
Proporciona descripciones detalladas y tutoriales acerca de cada uno de los 13 tipos de diagramas de UML 2.

`www-306.ibm.com/software/rational/offerings/design.html`
IBM proporciona información acerca del software de Rational disponible para el diseño de sistemas, y descargas de versiones de prueba de 30 días de varios productos, como IBM Rational Rose® XDE (Entorno de desarrollo extendido) Developer.

`www.embarcadero.com/products/describe/index.html`
Proporciona una licencia de prueba de 15 días para la herramienta de modelado de UML Describe™ de Embarcadero Technologies®.

`www.borland.com/together/index.html`
Proporciona una licencia gratuita de 30 días para descargar una versión de Borland® Together® Control-Center™: una herramienta de desarrollo de software que soporta el UML.

`www.ilogix.com/rhapsody/rhapsody.cfm`
Proporciona una licencia gratuita de 30 días para descargar una versión de prueba de I-Logix Rhapsody®: un entorno de desarrollo controlado por modelos y basado en UML 2.

`argouml.tigris.org`
Contiene información y descargas para ArgoUML, una herramienta gratuita de software libre de UML.

`www.objectsbydesign.com/books/booklist.html`
Provee una lista de libros acerca de UML y el diseño orientado a objetos.

`www.objectsbydesign.com/tools/umltools_byCompany.html`
Provee una lista de herramientas de software que utilizan UML, como IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody y Gentleware Poseidon para UML.

`www.ootips.org/ood-principles.html`
Proporciona respuestas a la pregunta "¿Qué se requiere para tener un buen diseño orientado a objetos?"

`www.cetus-links.org/oo_uml.html`
Introduce el UML y proporciona vínculos a numerosos recursos sobre UML.

Lecturas recomendadas

Los siguientes libros proporcionan información acerca del diseño orientado a objetos con el UML.

Ambler, S. *The Elements of the UML 2.0 Style.* Nueva York: Cambridge University Press, 2005.

Booch, G. *Object-Oriented Analysis and Design with Applications, Tercera edición*. Boston: Addison-Wesley, 2004.

Eriksson, H. *et al. UML 2 Toolkit*. Nueva York: John Wiley, 2003.

Kruchten, P. *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley, 2004.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design,* Segunda edición. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. *UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions*. Nueva York: John Wiley, 2004.

Rosenberg, D. y K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Rumbaugh, J., I. Jacobson y G. Booch. *The Complete UML Training Course.* Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

3.1 Suponga que habilitamos a un usuario de nuestro sistema ATM para transferir dinero entre dos cuentas bancarias. Modifique el diagrama de caso-uso de la figura 3.31 para reflejar este cambio.

3.2 Los _____ modelan las interacciones entre los objetos en un sistema, con énfasis acerca de *cuándo* ocurren estas interacciones.

a) Diagramas de clases
b) Diagramas de secuencia
c) Diagramas de comunicación
d) Diagramas de actividad

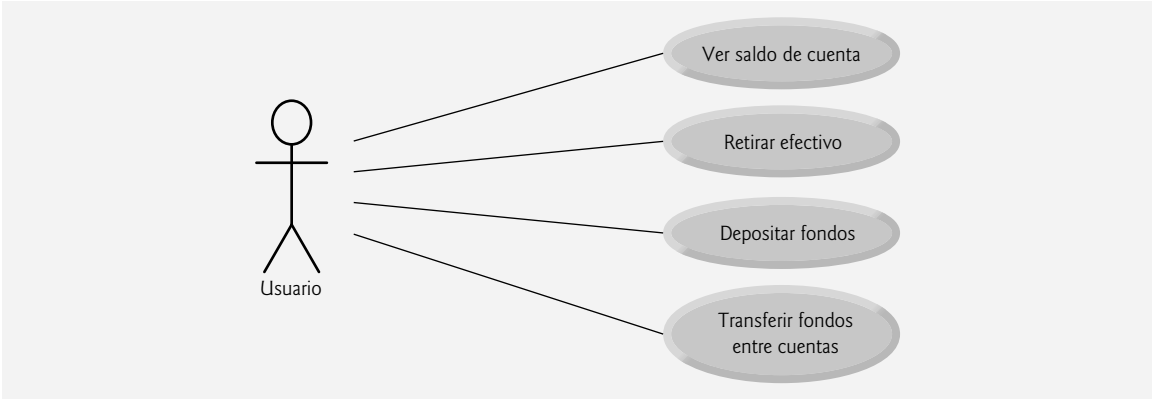

Figura 3.32 | Diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre varias cuentas.

3.3 ¿Cuál de las siguientes opciones lista las etapas de un ciclo de vida de software común en orden secuencial?
a) diseño, análisis, implementación, prueba
b) diseño, análisis, prueba, implementación
c) análisis, diseño, prueba, implementación
d) análisis, diseño, implementación, prueba

Respuestas a los ejercicios de autoevaluación del Caso de estudio de ingeniería de software

3.1 La figura 3.32 contiene un diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre cuentas.

3.2 b.

3.3 d.

3.11 Conclusión

En este capítulo aprendió muchas características importantes de C#, incluyendo: mostrar datos en la pantalla en un símbolo del sistema, introducir datos desde el teclado, realizar cálculos y tomar decisiones. Las aplicaciones que presentamos aquí lo introdujeron a los conceptos básicos de programación. Como verá en el capítulo 4, las aplicaciones de C# por lo general contienen sólo unas cuantas líneas de código en el método `Main`; estas instrucciones por lo común crean los objetos que realizan el trabajo de la aplicación. En el capítulo 4 aprenderá a implementar sus propias clases y a utilizar objetos de esas clases en aplicaciones.