

# 4

# Introducción a las clases y los objetos

## OBJETIVOS

En este capítulo aprenderá lo siguiente:

- Qué son las clases, los objetos, los métodos y las variables de instancia.
- Declarar una clase y utilizarla para crear un objeto.
- Implementar los comportamientos de una clase como métodos.
- Cómo implementar los atributos de una clase como variables de instancia y propiedades.
- Cómo llamar a los métodos de un objeto para que realicen sus tareas.
- Las diferencias entre las variables de instancia de una clase y las variables locales de un método.
- Utilizar un constructor para asegurar que los datos de un objeto se inicialicen cuando al crear el objeto.
- Las diferencias entre tipos por valor y tipos por referencia.

*Nada puede tener valor sin ser un objeto de utilidad.*

—Karl Marx

*Sus sirvientes públicos le sirven bien.*

—Adlai E. Stevenson

*Saber cómo responder a alguien que habla,  
Contestar a alguien que envía un mensaje.*

—Amenemope

*Usted verá algo nuevo.  
Dos cosas. Y las llamo:  
Cosa Uno y Cosa Dos.*

—Dr. Theodor Seuss Geisel

**Plan general**

- 4.1 Introducción
- 4.2 Clases, objetos, métodos, propiedades y variables de instancia
- 4.3 Declaración de una clase con un método e instanciamiento del objeto de una clase
- 4.4 Declaración de un método con un parámetro
- 4.5 Variables de instancia y propiedades
- 4.6 Diagrama de clases de UML con una propiedad
- 4.7 Ingeniería de software con propiedades y los descriptores de acceso `set` y `get`
- 4.8 Comparación entre tipos por valor y tipos por referencia
- 4.9 Inicialización de objetos con constructores
- 4.10 Los números de punto flotante y el tipo `decimal`
- 4.11 (Opcional) Caso de estudio de ingeniería de software: identificación de las clases en el documento de requerimientos del ATM
- 4.12 Conclusión

## 4.1 Introducción

En la sección 1.9 presentamos la terminología básica y los conceptos acerca de la programación orientada a objetos. En el capítulo 3 comenzó a utilizar esos conceptos para crear aplicaciones simples que mostraran mensajes al usuario, que obtuvieran información de él, realizaran cálculos y tomaran decisiones. Una característica común de todas las aplicaciones en el capítulo 3 fue que todas las instrucciones que realizaban tareas se encontraban en el método `Main`. Por lo general, las aplicaciones que usted desarrollará en este libro constarán de dos o más clases, y cada una contendrá uno o más métodos. Si se convierte en parte de un equipo de desarrollo en la industria, podría trabajar en aplicaciones que contengan cientos, o incluso hasta miles de clases. En este capítulo presentaremos un marco de trabajo simple para organizar las aplicaciones orientadas a objetos en C#.

Primero explicaremos el concepto de las clases mediante el uso de un ejemplo real. Después presentaremos cinco aplicaciones completas para demostrarle cómo crear y utilizar sus propias clases. Nuestro primer paso es comenzar el caso de estudio acerca de cómo desarrollar una clase tipo libro de calificaciones, que los instructores pueden utilizar para mantener las calificaciones de las pruebas de sus estudiantes. Durante los siguientes capítulos ampliaremos este caso de estudio y culminaremos con la versión que se presenta en el capítulo 8, Arreglos. El último ejemplo en este capítulo introduce el tipo `decimal` y lo utiliza para declarar cantidades monetarias dentro del contexto de una clase tipo cuenta bancaria, la cual mantiene el saldo de un cliente.

## 4.2 Clases, objetos, métodos, propiedades y variables de instancia

Comenzaremos con una analogía simple para ayudarle a comprender el concepto de las clases y su contenido. Suponga que desea conducir un auto y para hacer que aumente su velocidad debe presionar el pedal del acelerador. ¿Qué debe ocurrir antes de que pueda hacer esto? Bueno, antes de poder conducir un auto, alguien tiene que diseñarlo. Por lo general un auto empieza en forma de dibujos de ingeniería, similares a los planos de construcción que se utilizan para diseñar una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador, para que el auto aumente su velocidad. El pedal “oculta” los complejos mecanismos que se encargan de que el auto aumente su velocidad, de igual forma que el pedal del freno “oculta” los mecanismos que disminuyen la velocidad del auto y el volante “oculta” los mecanismos que hacen que el auto dé vuelta. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores puedan conducir un auto con facilidad.

Desafortunadamente, no puede conducir los dibujos de ingeniería de un auto. Antes de poder conducir un auto, éste debe construirse a partir de los dibujos de ingeniería que lo describen. Un auto completo tendrá un pedal acelerador verdadero para hacer que aumente su velocidad, pero aun así no es suficiente; el auto no acelerará por su propia cuenta, así que el conductor debe oprimir el pedal del acelerador.

Ahora utilizaremos nuestro ejemplo del auto para introducir los conceptos clave de programación de esta sección. Para realizar una tarea en una aplicación se requiere un método. El **método** describe los mecanismos que

se encargan de realizar sus tareas; y oculta al usuario las tareas complejas que realiza, de la misma forma que el pedal del acelerador de un auto oculta al conductor los complejos mecanismos para hacer que el auto vaya más rápido. En C# empezamos por crear una unidad de aplicación llamada **clase** para alojar (entre otras cosas) a un método, así como los dibujos de ingeniería de un auto alojan (entre otras cosas) el diseño del pedal del acelerador. En una clase se proporcionan uno o más métodos, que están diseñados para realizar las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener un método para depositar dinero, otro para retirar dinero y un tercer método para solicitar el saldo actual.

Así como no podemos conducir un dibujo de ingeniería de un auto, tampoco podemos “conducir” una clase. De la misma forma que alguien tiene que construir un auto a partir de sus dibujos de ingeniería para poder conducirlo, también debemos construir un **objeto** de una clase para poder hacer que una aplicación realice las tareas descritas por la clase. Ésta es una de las razones por las cuales C# se conoce como un lenguaje de programación orientado a objetos.

Cuando usted conduce un auto, si opriime el pedal del acelerador se envía un mensaje al auto para que realice una tarea: hacer que el auto vaya más rápido. De manera similar, se envían **mensajes** a un objeto; a cada mensaje se le conoce como la **llamada a un método** e indica a un método del objeto que realice su tarea.

Hasta ahora hemos utilizado la analogía del auto para introducir las clases, los objetos y los métodos. Además de las capacidades con las que cuenta un auto, también tiene muchos **atributos** como su color, el número de puertas, la cantidad de gasolina en su tanque, su velocidad actual y el total de kilómetros recorridos (es decir, la lectura de su odómetro). Al igual que las capacidades del auto, estos atributos se representan como parte del diseño del auto en sus diagramas de ingeniería. Cuando usted conduce un auto, estos atributos siempre están asociados con él. Cada auto mantiene sus propios atributos. Por ejemplo, cada auto sabe cuánta gasolina tiene en su propio tanque, pero no cuánta hay en los tanques de otros autos. De manera similar, un objeto tiene atributos que lleva consigo cuando se utiliza en una aplicación. Estos atributos se especifican como parte de la clase del objeto. Por ejemplo, un objeto cuenta bancaria tiene un atributo llamado saldo, que representa la cantidad de dinero en la cuenta. Cada objeto cuenta bancaria conoce el saldo en la cuenta que representa, pero no los saldos de las otras cuentas en el banco. Los atributos se especifican mediante las **variables de instancia** de la clase.

Hay que tener en cuenta que no es necesario tener acceso directo a estos atributos. El fabricante de autos no desea que los conductores desarmen el motor del auto para observar la cantidad de gasolina en su tanque. En vez de ello, el conductor puede revisar el medidor en el tablero. El banco no desea que sus clientes entren a la bóveda para contar la cantidad de dinero en una cuenta. En vez de ello, los clientes van con un cajero en el banco. De manera similar, usted no necesita tener acceso a las variables de instancia de un objeto para poder utilizarlas. Puede utilizar las **propiedades** de un objeto. Las propiedades contienen **descriptores de acceso get** para leer los valores de las variables y **descriptores de acceso set** para almacenar valores en ellas.

El resto de este capítulo contiene ejemplos que demuestran los conceptos que presentamos aquí, dentro del contexto de la analogía del auto. Los primeros cuatro ejemplos que se sintetizan a continuación, se encargan de construir en forma incremental una clase llamada **LibroCalificaciones**:

1. El primer ejemplo presenta una clase llamada **LibroCalificaciones** con un método que sólo muestra un mensaje de bienvenida cuando se le llama. Le mostraremos cómo **crear un objeto** de esa clase y cómo llamarlo para que muestre el mensaje de bienvenida.
2. El segundo ejemplo modifica el primero, al permitir que el método reciba el nombre de un curso como “argumento” y al mostrar ese nombre como parte del mensaje de bienvenida.
3. El tercer ejemplo muestra cómo almacenar el nombre del curso en un objeto tipo **LibroCalificaciones**. Para esta versión de la clase, también le mostraremos cómo utilizar las propiedades para establecer el nombre del curso y obtenerlo.
4. El cuarto ejemplo demuestra cómo pueden inicializarse los datos en un objeto tipo **LibroCalificaciones**, a la hora de crear el objeto; el constructor de la clase se encarga de realizar el proceso de inicialización.

El último ejemplo en el capítulo presenta una clase llamada **Cuenta**, la cual refuerza los conceptos presentados en los primeros cuatro ejemplos e introduce el tipo **decimal**; un número **decimal** puede contener un punto decimal, como en 0.0345, -7.23 y 100.7, y se utiliza para realizar cálculos precisos, en especial los que implican valores monetarios. Para este fin presentamos una clase llamada **Cuenta**, la cual representa una cuenta bancaria

y mantiene su saldo `decimal`. La clase contiene un método para acreditar un depósito a la cuenta, con lo cual se incrementa el saldo; y una propiedad para obtener el saldo y asegurarse de que todos los valores asignados al mismo no sean negativos. El constructor de la clase inicializa el saldo de cada objeto tipo `Cuenta`, a la hora de crear el objeto. Crearemos dos objetos tipo `Cuenta` y haremos depósitos en cada uno de ellos para mostrar que cada objeto mantiene su propio saldo. El ejemplo también demuestra cómo introducir e imprimir en pantalla números tipo `decimal`.

### 4.3 Declaración de una clase con un método e instanciamiento del objeto de una clase

Empezaremos con un ejemplo que consiste en las clases `LibroCalificaciones` (figura 4.1) y `PruebaLibroCalificaciones` (figura 4.2). La clase `LibroCalificaciones` (declarada en el archivo `LibroCalificaciones.cs`) se utilizará para mostrar un mensaje en la pantalla (figura 4.2) para dar la bienvenida, al instructor, a la aplicación del libro de calificaciones. La clase `PruebaLibroCalificaciones` (declarada en el archivo `PruebaLibroCalificaciones.cs`) es una clase de prueba en la que el método `Main` creará y utilizará un objeto de la clase `LibroCalificaciones`. Por convención declaramos a las clases `LibroCalificaciones` y `PruebaLibroCalificaciones` en archivos separados, de tal forma que el nombre de cada archivo concuerde con el nombre de la clase que contiene.

Para empezar, seleccione **Archivo > Nuevo proyecto...** para abrir el cuadro de diálogo **Nuevo proyecto**, y después cree una **Aplicación de consola** llamada `LibroCalificaciones`. Elimine todo el código que el IDE proporciona de manera automática y sustitúyalo con el código de la figura 4.1.

#### *Clase LibroCalificaciones*

La **declaración de la clase** `LibroCalificaciones` (figura 4.1) contiene un método llamado `MostrarMensaje` (líneas 8-11), que muestra un mensaje en la pantalla. La línea 10 de la clase muestra el mensaje. Recuerde que una clase es como un plano de construcción; necesitamos crear un objeto de esta clase y llamar a su método para hacer que se ejecute la línea 10 y muestre su mensaje (haremos esto en la figura 4.2).

La declaración de la clase comienza en la línea 5. La palabra clave `public` es un **modificador de acceso**. Por ahora, a todas las clases las declararemos como `public`. La declaración de cada clase contiene la palabra clave `class`, seguida del nombre de la clase. El cuerpo de cada clase va encerrado entre un par de llaves izquierda y derecha (`{` y `}`), como en las líneas 6 y 12 de `LibroCalificaciones`.

En el capítulo 3, cada clase que declaramos tenía un método llamado `Main`. La clase `LibroCalificaciones` también tiene un método: `MostrarMensaje` (líneas 8-11). Recuerde que `Main` es un método especial que siempre se llama de manera automática cuando se ejecuta una aplicación. La mayoría de los métodos no se llama de manera automática. Como pronto verá, deberá llamar al método `MostrarMensaje` para indicarle que realice su tarea.

La declaración del método empieza con la palabra clave `public` para indicar que el método está “disponible para el público”; es decir, los métodos de otras clases pueden llamar a este método desde fuera del cuerpo de la

```

1 // Fig. 4.1: LibroCalificaciones.cs
2 // Declaración de una clase con un método.
3 using System;
4
5 public class LibroCalificaciones
6 {
7     // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
8     public void MostrarMensaje()
9     {
10         Console.WriteLine( "¡Bienvenido al Libro de calificaciones!" );
11     } // fin del método MostrarMensaje
12 } // fin de la clase LibroCalificaciones

```

Figura 4.1 | Declaración de la clase con un método.

declaración de la clase. La palabra clave **void** (conocida como el **tipo de valor de retorno** del método) indica que este método no devolverá (o regresará) ninguna información al **método que lo llamó** cuando complete su tarea. Cuando se hace una llamada a un método que especifica un tipo de valor de retorno distinto de **void** y completa su tarea, el método devuelve un resultado al método que lo llamó. Por ejemplo, cuando usted va a un cajero automático (ATM) y solicita el saldo de su cuenta, espera que el ATM le devuelva un valor que represente su saldo. Si tiene un método llamado **Cuadrado** que devuelve el cuadrado de su argumento, sería de esperarse que la instrucción

```
int resultado = Cuadrado( 2 );
```

devolviera el valor 4 del método **Cuadrado** y lo asignara a la variable **resultado**. Si tiene un método llamado **Maximo** que devuelva el mayor de los tres argumentos tipo entero, sería de esperarse que la instrucción

```
int mayor = Maximo( 27, 114, 51 );
```

devolviera el valor 114 del método **Maximo** y asignara ese valor a la variable **mayor**. Ya hemos utilizado métodos que devuelven información; por ejemplo, en el capítulo 3 utilizamos el **método ReadLine de Console** para introducir una cadena escrita por el usuario mediante el teclado. Cuando **ReadLine** recibe un valor como entrada, devuelve ese valor para utilizarlo en la aplicación.

El nombre del método **MostrarMensaje** va después del tipo de valor de retorno (línea 8). Por convención, los nombres de los métodos empiezan con la primera letra en minúscula, y todas las palabras subsiguientes en ese nombre empiezan con letra mayúscula. Los paréntesis después del nombre del método indican que éste es un método. Un conjunto vacío de paréntesis, como el que se muestra en la línea 8, indica que este método no requiere información adicional para realizar su tarea. Por lo general, a la línea 8 se le conoce como el **encabezado del método**. El cuerpo de cada método está delimitado por las llaves izquierda y derecha, como en las líneas 9 y 11.

El cuerpo de un método contiene una o varias instrucciones que realizan la tarea de ese método. En este caso, el método contiene una instrucción (línea 10) que muestra el mensaje "**¡Bienvenido al libro de calificaciones!**", seguido de una nueva línea en la ventana de consola. Después de que se ejecuta esta instrucción, el método completa su tarea.

A continuación nos gustaría utilizar la clase **LibroCalificaciones** en una aplicación. Como aprendió en el capítulo 3, el método **Main** comienza la ejecución de toda aplicación. La clase **LibroCalificaciones** no puede comenzar una aplicación, ya que no contiene a **Main**. Esto no fue un problema en el capítulo 3, ya que todas las clases que declaramos tenían un método **Main**. Para corregir este problema para la clase **LibroCalificaciones**, debemos declarar una clase separada que contenga un método **Main** o colocar un método **Main** en la clase **LibroCalificaciones**. Para ayudarlo en su preparación para las aplicaciones más extensas con las que se encontrará más adelante en este libro y en la industria, utilizaremos una clase separada (en este ejemplo, **PruebaLibroCalificaciones**) que contiene el método **Main** para probar cada una de las nuevas clases que vayamos a crear en este capítulo.

### **Agregar una clase a un proyecto de Visual C#**

Para cada uno de los ejemplos en este capítulo, agregará una clase a su aplicación de consola. Para ello, haga clic con el botón derecho en el nombre del proyecto dentro del **Explorador de soluciones** y seleccione **Agregar > Nuevo elemento...** del menú desplegable. En el cuadro de diálogo **Agregar nuevo elemento** que aparezca, seleccione **Archivo de código** y escriba el nombre de su nuevo archivo; en este caso, **PruebaLibroCalificaciones.cs**. A continuación se agregará un nuevo archivo en blanco a su proyecto. Agregue el código de la figura 4.2 a este archivo.

### **Clase PruebaLibroCalificaciones**

La declaración de la clase **PruebaLibroCalificaciones** (figura 4.2) contiene el método **Main** que controla la ejecución de nuestra aplicación. Cualquier clase que contenga un método **Main** (como se muestra en la línea 7) puede utilizarse para ejecutar una aplicación. La declaración de esta clase comienza en la línea 4 y termina en la 15. La clase sólo contiene un método **Main**, que es común en muchas clases que sólo se utilizan para iniciar la ejecución de una aplicación.

Las líneas 7-14 declaran el método **Main**. Una parte clave de hacer que el método **Main** inicie la ejecución de la aplicación es la palabra clave **static** (línea 7), que establece que **Main** es un método estático. Un método

```

1 // Fig. 4.2: PruebaLibroCalificaciones.cs
2 // Crea un objeto LibroCalificaciones y llama a su método MostrarMensaje.
3
4 public class PruebaLibroCalificaciones
5 {
6     // El método Main comienza la ejecución del programa
7     public static void Main( string[] args )
8     {
9         // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
10        LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
11
12        // llama al método MostrarMensaje de miLibroCalificaciones
13        miLibroCalificaciones.MostrarMensaje();
14    } // fin de Main
15 } // fin de la clase PruebaLibroCalificaciones

```

¡Bienvenido al Libro de calificaciones!

**Figura 4.2** | Crear un objeto `LibroCalificaciones` y llamar a su método `MostrarMensaje`.

`static` es especial, ya que puede llamarse sin necesidad de crear primero un objeto de la clase (en este caso, `PruebaLibroCalificaciones`) en la que está declarado. En el capítulo 7, Métodos: un análisis más detallado, explicaremos la función de los métodos `static`.

En esta aplicación queremos llamar al método `MostrarMensaje` de la clase `LibroCalificaciones` para mostrar el mensaje de bienvenida en la ventana de consola. Por lo general, no se puede llamar a un método que pertenezca a otra clase, sino hasta que se cree un objeto de esa clase, como se muestra en la línea 10. Para empezar, declaramos la variable `miLibroCalificaciones`. Observe que el tipo de la variable es `LibroCalificaciones`: la clase que declaramos en la figura 4.1. Cada nueva clase que usted crea se convierte en un nuevo tipo en C#, el cual puede utilizarse para declarar variables y crear objetos. Los nuevos tipos de clases son accesibles para todas las clases dentro del mismo proyecto. Puede declarar nuevos tipos de clases según lo requiera; ésta es una de las razones por las que C# se conoce como un *lenguaje extensible*.

La variable `miLibroCalificaciones` (línea 10) se inicializa con el resultado de la *expresión de creación de objeto* `new LibroCalificaciones()`. El operador `new` crea un nuevo objeto de la clase especificada a la derecha de la palabra clave (es decir, `LibroCalificaciones`). Los paréntesis a la derecha de `LibroCalificaciones` son requeridos. Como aprenderá en la sección 4.9, esos paréntesis en combinación con el nombre de una clase representan una llamada a un constructor, que es similar a un método pero se utiliza sólo cuando se crea el objeto, para inicializar sus datos. En esta sección veremos que pueden colocarse los datos entre paréntesis para especificar los valores iniciales para los datos del objeto. Por ahora, sólo dejaremos los paréntesis vacíos.

Podemos utilizar a `miLibroCalificaciones` para llamar a su método `MostrarMensaje`. La línea 13 llama al método `MostrarMensaje` (líneas 8-11 de la figura 4.1) mediante el uso de la variable `miLibroCalificaciones`, seguida de un *operador punto* (`.`), el nombre del método `MostrarMensaje` y un conjunto vacío de paréntesis. Esta llamada hace que el método `MostrarMensaje` realice su tarea. La llamada a este método es distinta a las llamadas a los métodos en el capítulo 3 que mostraban información en una ventana de consola; cada una de esas llamadas a métodos proporcionaban argumentos que especificaban los datos a mostrar. Al principio de la línea 13, “`miLibroCalificaciones`”, indica que `Main` debe utilizar el objeto `LibroCalificaciones` que se creó en la línea 10. Los paréntesis vacíos en la línea 8 de la figura 4.1 indican que el método `MostrarMensaje` no requiere información adicional para realizar su tarea. Por esta razón, la llamada al método (línea 13 de la figura 4.2) especifica un conjunto vacío de paréntesis después del nombre del método, para indicar que no se van a pasar argumentos al método `MostrarMensaje`. Cuando `MostrarMensaje` completa su tarea, el método `Main` continúa su ejecución en la línea 14. Como éste es el final del método `Main`, la aplicación termina.

#### **Diagrama de clases de UML para la clase `LibroCalificaciones`**

La figura 4.3 presenta un *diagrama de clases de UML* para la clase `LibroCalificaciones` de la figura 4.1. Recuerde que en la sección 1.9 vimos que UML es un lenguaje gráfico, utilizado por los programadores para



**Figura 4.3** | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene una operación `public` llamada `MostrarMensaje`.

representar sus sistemas orientados a objetos de una manera estandarizada. En UML, cada clase se modela en un diagrama de clases en forma de un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado en forma horizontal y en negrita. El compartimiento de en medio contiene los atributos de la clase, que en C# corresponden a las variables de instancia y las propiedades. En la figura 4.3 el compartimiento de en medio está vacío, ya que la versión de la clase `LibroCalificaciones` en la figura 4.1 no tiene atributos. El compartimiento inferior contiene las operaciones de la clase, que en C# corresponden a los métodos. Para modelar las operaciones, UML lista el nombre de la operación seguido de un conjunto de paréntesis. La clase `LibroCalificaciones` tiene un método llamado `MostrarMensaje`, por lo que el compartimiento inferior de la figura 4.3 lista una operación con este nombre. El método `MostrarMensaje` no requiere información adicional para realizar sus tareas, por lo cual hay paréntesis vacíos después de `MostrarMensaje` en el diagrama de clases, de igual forma que como aparecieron en la declaración del método en la línea 8 de la figura 4.1. El signo más (+) que va antes del nombre de la operación indica que `MostrarMensaje` es una operación pública en UML (es decir, un método `public` en C#). Al signo más también se le conoce como **símbolo de visibilidad pública**. A menudo utilizaremos los diagramas de clases de UML para sintetizar los atributos y las operaciones de una clase.

## 4.4 Declaración de un método con un parámetro

En nuestra analogía del auto de la sección 4.2, hablamos sobre el hecho de que al oprimir el pedal del acelerador se envía un mensaje al auto para que realice una tarea: hacer que vaya más rápido. Pero, ¿qué tan rápido debería acelerar el auto? Como sabe, entre más oprima el pedal, mayor será la aceleración del auto. Por lo tanto, el mensaje para el auto en realidad incluye tanto la tarea a realizar como información adicional que ayuda al auto a realizar su tarea. A la información adicional se le conoce como **parámetro**; el valor del parámetro ayuda al auto a determinar qué tan rápido debe acelerar. De manera similar, un método puede requerir uno o más parámetros que representan la información adicional que necesita para realizar su tarea. La llamada a un método proporciona valores (llamados argumentos) para cada uno de los parámetros de ese método. Por ejemplo, el método `Console.WriteLine` requiere un argumento que especifica los datos a mostrar en una ventana de consola. Así mismo, para realizar un depósito en una cuenta bancaria, un método llamado `Deposito` especifica un parámetro que representa el monto a depositar. Cuando se hace una llamada al método `Deposito`, se asigna al parámetro del método un valor como argumento que representa el monto a depositar. Entonces el método realiza un depósito por ese monto e incrementa el balance de la cuenta.

Nuestro siguiente ejemplo declara la clase `LibroCalificaciones` (figura 4.4) con un método `MostrarMensaje` que señala el nombre del curso como parte del mensaje de bienvenida (en la figura 4.5 podrá ver la ejecución de ejemplo). El nuevo método `MostrarMensaje` requiere un parámetro que representa el nombre del curso a imprimir en pantalla.

Antes de hablar sobre las nuevas características de la clase `LibroCalificaciones`, veamos cómo se utiliza la nueva clase desde el método `Main` de la clase `PruebaLibroCalificaciones` (figura 4.5). La línea 12 crea un objeto de la clase `LibroCalificaciones` y lo asigna a la variable `miLibroCalificaciones`. La línea 15 pide al usuario que escriba el nombre de un curso. La línea 16 lee el nombre que introduce el usuario y lo asigna a la variable `nombreDelCurso`, mediante el uso del método `ReadLine` de `Console` para realizar la operación de entrada. El usuario escribe el nombre del curso y oprime `Intro` para enviarlo a la aplicación. Observe que al oprimir `Intro` se inserta un carácter de nueva línea al final de los caracteres escritos por el usuario. El método `ReadLine` lee los caracteres que escribió el usuario hasta encontrar el carácter de nueva línea y después devuelve un valor tipo `string` que contiene los caracteres hasta, pero sin incluir a, la nueva línea. El carácter de nueva línea se descarta.

```

1 // Fig. 4.4: LibroCalificaciones.cs
2 // Declaración de la clase con un método que tiene un parámetro.
3 using System;
4
5 public class LibroCalificaciones
6 {
7     // muestra un mensaje de bienvenida para el usuario del LibroCalificaciones
8     public void MostrarMensaje( string nombreCurso )
9     {
10         Console.WriteLine( "¡Bienvenido al libro de calificaciones para\n{0}!", 
11             nombreCurso );
12     } // fin del método MostrarMensaje
13 } // fin de la clase LibroCalificaciones

```

**Figura 4.4** | Declaración de la clase con un método que tiene un parámetro.

```

1 // Fig. 4.5: LibroPruebaCalificaciones.cs
2 // Crea objeto LibroCalificaciones y pasa una cadena a
3 // su método MostrarMensaje.
4 using System;
5
6 public class LibroPruebaCalificaciones
7 {
8     // El método Main comienza la ejecución del programa
9     public static void Main( string[] args )
10    {
11        // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
12        LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
13
14        // pide el nombre del curso y lo recibe como entrada
15        Console.WriteLine( "Por favor escriba el nombre del curso:" );
16        string nombreDelCurso = Console.ReadLine(); // lee una línea de texto
17        Console.WriteLine(); // imprime en pantalla una línea en blanco
18
19        // llama al método MostrarMensaje de miLibroCalificaciones
20        // y pasa nombreDelCurso como argumento
21        miLibroCalificaciones.MostrarMensaje( nombreDelCurso );
22    } // fin de Main
23 } // fin de la clase LibroPruebaCalificaciones

```

Por favor escriba el nombre del curso:  
**CS101 Introducción a la programación en C#**

¡Bienvenido al libro de calificaciones para  
**CS101 Introducción a la programación en C#!**

**Figura 4.5** | Creación de un objeto LibroCalificaciones y paso de una cadena a su método MostrarMensaje.

La línea 21 llama al método `MostrarMensaje` de `miLibroCalificaciones`. La variable `nombreDelCurso` entre paréntesis es el argumento que se pasa al método `MostrarMensaje` para que pueda realizar su tarea. El valor de la variable `nombreDelCurso` en `Main` se convierte en el valor del parámetro `nombreCurso` del método `MostrarMensaje` en la línea 8 de la figura 4.4. Al ejecutar esta aplicación, observe que el método `MostrarMensaje` imprime en pantalla el nombre que usted escribió como parte del mensaje de bienvenida (figura 4.5).



#### Observación de ingeniería de software 4.1

*Por lo general, los objetos se crean mediante el uso de new. Una excepción es la literal de cadena que está encerrada entre comillas, tal como "hola". Las literales de cadena son referencias a objetos string que C# crea de manera implícita.*

### Más sobre los argumentos y los parámetros

Al declarar un método, debe especificar en su declaración si éste requiere datos para realizar su tarea. Para ello hay que colocar información adicional en la **lista de parámetros** del método, la cual se encuentra en los paréntesis que van después del nombre del método. La lista de parámetros puede contener cualquier número de parámetros, incluso ninguno. Los paréntesis vacíos después del nombre del método (como en la figura 4.1, línea 8) indican que un método no requiere parámetros. En la figura 4.4, la lista de parámetros de `MostrarMensaje` (línea 8) declara que el método requiere un parámetro. Cada parámetro debe especificar un tipo y un identificador. En este caso, el tipo `string` y el identificador `nombreCurso` indican que el método `MostrarMensaje` requiere un objeto `string` para realizar su tarea. En el instante en que se llama al método, el valor del argumento en la llamada se asigna al parámetro correspondiente (en este caso, `nombreCurso`) en el encabezado del método. Después, el cuerpo del método utiliza el parámetro `nombreCurso` para acceder al valor. Las líneas 10-11 de la figura 4.4 muestran el valor del parámetro `nombreCurso`, mediante el uso del elemento de formato `{0}` en el primer argumento de `WriteLine`. Observe que el nombre de la variable de parámetro (figura 4.4, línea 8) puede ser igual o distinto al nombre de la variable de argumento (figura 4.5, línea 21).

Un método puede especificar múltiples parámetros; sólo hay que separar un parámetro de otro mediante una coma. El número de argumentos en la llamada a un método debe concordar con el número de parámetros en la lista de parámetros de la declaración del método que se llamó. Además, los tipos de los argumentos en la llamada a un método deben ser consistentes con los tipos de los parámetros correspondientes en la declaración del método (como veremos en capítulos posteriores, no siempre se requiere que el tipo de un argumento y el tipo de su correspondiente parámetro sean idénticos). En nuestro ejemplo, la llamada al método pasa un argumento de tipo `string` (`nombreDelCurso` se declara como `string` en la línea 16 de la figura 4.5) y la declaración del método especifica un parámetro de tipo `string` (línea 8 en la figura 4.4). Por lo tanto, el tipo del argumento en la llamada al método concuerda exactamente con el tipo del parámetro en el encabezado del método.



### Error común de programación 4.1

*Si el número de argumentos en la llamada a un método no concuerda con el número de parámetros en la declaración del método, se produce un error de compilación.*



### Error común de programación 4.2

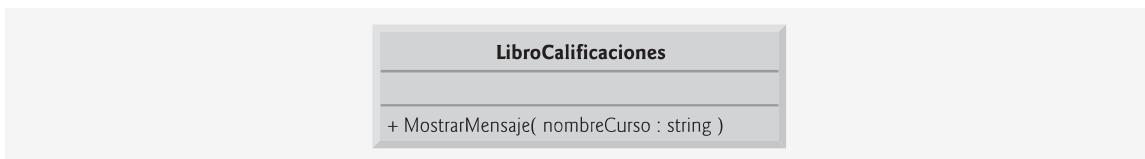
*Si los tipos de los argumentos en la llamada a un método no son consistentes con los tipos de los parámetros correspondientes en la declaración del método, se produce un error de compilación.*

### Diagrama de clases actualizado para la clase LibroCalificaciones

El diagrama de clases de UML de la figura 4.6 modela la clase `LibroCalificaciones` de la figura 4.4. Al igual que la figura 4.4, esta clase `LibroCalificaciones` contiene la operación `public` llamada `MostrarMensaje`. Sin embargo, esta versión de `MostrarMensaje` tiene un parámetro. La forma en que UML modela un parámetro es un poco distinta a la de C#, ya que lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre paréntesis, después del nombre de la operación. UML cuenta con varios tipos de datos que son similares a los tipos de C#. Por ejemplo, los tipos `String` e `Integer` de UML corresponden a los tipos `string` e `int` de C#. Por desgracia, UML no proporciona tipos que correspondan a todos los tipos de C#. Por esta razón y para evitar confusiones entre los tipos de UML y los de C#, sólo utilizamos tipos de C# en nuestros diagramas de UML. El método `MostrarMensaje` de la clase `LibroCalificaciones` (figura 4.4) tiene un parámetro `string` llamado `nombreCurso`, por lo que en la figura 4.6 se lista a `nombreCurso : string` entre los paréntesis que van después de `MostrarMensaje`.

### Observaciones acerca del uso de las directivas

Observe la directiva `using` en la figura 4.5 (línea 4). Esto indica al compilador que la aplicación utiliza clases en el espacio de nombres `System`, como la clase `Console`. ¿Por qué necesitamos una directiva `using` para utilizar la clase `Console`, pero no la clase `LibroCalificaciones`? Existe una relación especial entre las clases que se compilan en el mismo proyecto, como las clases `LibroCalificaciones` y `PruebaLibroCalificaciones`. De manera predeterminada se considera que dichas clases se encuentran en el mismo espacio de nombres. No se requiere una directiva `using` cuando una clase en un espacio de nombres utiliza a otra dentro del mismo espacio de nombres;



**Figura 4.6** | Diagrama de clases de UML que indique que la clase `LibroCalificaciones` tiene una operación pública llamada `MostrarMensaje`, con un parámetro llamado `nombreCurso` de tipo `string`.

como cuando la clase `PruebaLibroCalificaciones` utiliza a la clase `LibroCalificaciones`. En la sección 9.14 verá cómo puede declarar sus propios espacios de nombres mediante la palabra clave `namespace`. Por cuestión de simplicidad, nuestros ejemplos en este capítulo no declaran un espacio de nombres. Cualquier clase que no se coloque de manera explícita en un espacio de nombres se coloca de manera implícita en lo que se denomina **espacio de nombres global**.

En realidad no se requiere la directiva `using` en la línea 4 si nos referimos siempre a la clase `Console` como `System.Console`, con lo cual se incluye el espacio de nombres y el nombre de la clase completos. A esto se le conoce como el **nombre de clase completamente calificado**. Por ejemplo, podríamos escribir la línea 15 como

```
System.Console.WriteLine( "Por favor escriba el nombre del curso:" );
```

La mayoría de los programadores de C# considera que el uso de nombres completamente calificados es incómodo, por lo que prefieren utilizar directivas `using`. El código que genera el Diseñador de formularios de Visual C# utiliza nombres completamente calificados.

## 4.5 Variables de instancia y propiedades

En el capítulo 3 declaramos todas las variables de una aplicación en el método `Main` de esa aplicación. Las variables que se declaran en el cuerpo de un método específico se conocen como **variables locales** y sólo se pueden utilizar en ese método. Cuando termina un método, se pierden los valores de sus variables locales. En la sección 4.2 vimos que un objeto tiene atributos que lleva consigo cuando se utiliza en una aplicación. Dichos atributos existen antes de que se llame a un método de un objeto, y después de que el método completa su ejecución.

En la declaración de una clase, los atributos se representan como variables. A dichas variables se les denomina **campos** y se declaran dentro de la declaración de una clase, pero fuera de los cuerpos de las declaraciones de los métodos de esa clase. Cuando cada objeto de una clase mantiene su propia copia de un atributo, al campo que representa ese atributo se le denomina variable de instancia; cada objeto (instancia de la clase) tiene una instancia separada de esa variable en la memoria. [Nota: en el capítulo 9, Clases y objetos: un análisis más detallado, hablaremos sobre otro tipo de campo llamado variable `static`, en donde todos los objetos de la misma clase comparten una copia de la variable.]

Por lo general, una clase consiste de una o más propiedades que manipulan los atributos que pertenecen a un objeto específico de la clase. El ejemplo en esta sección demuestra una clase `LibroCalificaciones` que contiene una variable de instancia llamada `nombreCurso` para representar el nombre del curso de un objeto `LibroCalificaciones` específico, y una propiedad llamada `NombreCurso` para manipular a `nombreCurso`.

### La clase `LibroCalificaciones` con una variable de instancia y una propiedad

En nuestra siguiente aplicación (figuras 4.7-4.8), la clase `LibroCalificaciones` (figura 4.7) mantiene el nombre del curso como una variable de instancia, de manera que se pueda utilizar o modificar en cualquier momento, durante la ejecución de una aplicación. La clase también contiene un método llamado `MostrarMensaje` (líneas 24-30) y una propiedad llamada `NombreCurso` (líneas 11-21). En el capítulo 2 vimos que las propiedades se utilizan para manipular los atributos de un objeto. Por ejemplo, en ese capítulo utilizamos la propiedad `Text` de un objeto `Label` para especificar el texto a mostrar en el objeto `Label`. En este ejemplo utilizamos una propiedad en el código, en vez de usarla en la ventana `Propiedades` del IDE. Para ello, primero declaramos una propiedad como miembro de la clase `LibroCalificaciones`. Como pronto verá, la propiedad `NombreCurso` de `LibroCalificaciones` puede utilizarse para almacenar el nombre de un curso en un objeto `LibroCalificaciones` (en la variable de instancia `nombreCurso`), o para recuperar el nombre del curso del objeto `LibroCalificaciones` (de

```

1 // Fig. 4.7: LibroCalificaciones.cs
2 // Clase LibroCalificaciones que contiene una variable de instancia cursoNombre
3 // y una propiedad para obtener (get) y establecer (set) su valor.
4 using System;
5
6 public class LibroCalificaciones
7 {
8     private string nombreCurso; // nombre del curso para este LibroCalificaciones
9
10    // propiedad para obtener (get) y establecer (set) el nombre del curso
11    public string NombreCurso
12    {
13        get
14        {
15            return nombreCurso;
16        } // fin de get
17        set
18        {
19            nombreCurso = value;
20        } // fin de set
21    } // fin de la propiedad NombreCurso
22
23    // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
24    public void MostrarMensaje()
25    {
26        // usa la propiedad NombreCurso para obtener el
27        // nombre del curso que representa este LibroCalificaciones
28        Console.WriteLine("¡Bienvenido al libro de calificaciones para\n{0}!", 
29                         NombreCurso); // muestra la propiedad NombreCurso
30    } // fin del método MostrarMensaje
31 } // fin de la clase LibroCalificaciones

```

**Figura 4.7** | La clase `LibroCalificaciones` contiene una variable de instancia `private` llamada `nombreCurso` y una propiedad `public` para obtener (`get`) y establecer (`set`) su valor.

la variable de instancia `nombreCurso`). El método `MostrarMensaje` (que ahora no especifica parámetros) sigue mostrando un mensaje de bienvenida, que incluye el nombre del curso. No obstante, ahora el método utiliza la propiedad `NombreCurso` para obtener el nombre del curso de la variable de instancia `nombreCurso`.

Un instructor común enseña más de un curso, cada uno con su propio nombre. La línea 8 que declara a `nombreCurso` como una variable de tipo `string`, es una declaración de una variable de instancia, ya que la variable se declara dentro del cuerpo de la clase (líneas 7-31), pero fuera de los cuerpos del método (líneas 24-30) y de la propiedad (líneas 11-21) de la clase. Cada instancia (es decir, objeto) de la clase `LibroCalificaciones` contiene una copia de cada una de las variables de instancia. Por ejemplo, si hay dos objetos `LibroCalificaciones`, cada uno tiene su propia copia de `nombreCurso` (una por objeto). Todos los métodos y las propiedades de la clase `LibroCalificaciones` pueden manipular en forma directa su variable de instancia `nombreCurso`, pero se considera una buena práctica que los métodos de una clase utilicen las propiedades de la misma para manipular las variables de instancia (como se hace en la línea 29 del método `MostrarMensaje`). Pronto se volverán más claras las razones de hacer esto, en relación con la ingeniería de software.

### Modificadores de acceso `public` y `private`

La mayoría de las declaraciones de las variables de instancia va precedida por la palabra clave `private` (como en la línea 8). Al igual que `public`, la palabra clave `private` es un modificador de acceso. Las variables o los métodos que se declaran con el modificador de acceso `private` son accesibles sólo para los métodos de la clase en la que están declarados. Por ende, la variable `nombreCurso` sólo se puede utilizar dentro de la propiedad `NombreCurso` y del método `MostrarMensaje` de la clase `LibroCalificaciones`.



### Observación de ingeniería de software 4.2

*Coloque un modificador de acceso antes de cada campo y dé la declaración de cada método. Como regla general, las variables de instancia deben declararse como `private`; los métodos y las propiedades como `public`. Si se omite el modificador de acceso antes del miembro de una clase, el miembro se declara de manera implícita como `private`. (Más adelante veremos que es apropiado declarar ciertos métodos `private`, sólo si otros métodos de la clase tendrán acceso a ellos.)*



### Buena práctica de programación 4.1

*Es preferible listar primero los campos de una clase para que, cuando lea el código, pueda ver los nombres y los tipos de las variables antes de que se utilicen en los métodos de la clase. Es posible listar los campos de la clase en cualquier parte de ésta que sea fuera de las declaraciones de sus métodos, pero será más fácil leer el código si se agrupan en un solo lugar.*



### Buena práctica de programación 4.2

*Al colocar una línea en blanco entre las declaraciones de los métodos y de las propiedades, se mejora la legibilidad de la aplicación.*

Al proceso de declarar las variables de instancia con el modificador de acceso `private` se le conoce como **ocultamiento de información**. Cuando una aplicación crea (instancia) un objeto de la clase `LibroCalificaciones`, la variable `nombreCurso` se encapsula (oculta) en el objeto y sólo los métodos y las propiedades de la clase de ese objeto pueden tener acceso a ella. En la clase `LibroCalificaciones`, la propiedad `NombreClase` manipula la variable de instancia `nombreCurso`.

#### Cómo establecer y obtener los valores de las variables de instancia `private`

¿Cómo podemos permitir que un programa manipule las variables de instancia `private` de una clase y asegurar que permanezcan en un estado válido? Necesitamos proporcionar medios controlados para que los programadores puedan “obtener” (es decir, recuperar) y “establecer” (esto es, modificar) el valor en una variable de instancia. Para estos fines, los programadores que utilizan lenguajes distintos a C# utilizan por lo general métodos conocidos como `get` y `set`. Estos métodos casi siempre son `public` y proporcionan los medios para que el cliente acceda a los datos `private` o los modifique. Por cuestión de historia, estos métodos comienzan con las palabras “Get” (“Obtener”) y “Set” (“Establecer”); por ejemplo, en nuestra clase `LibroCalificaciones`, si quisieramos utilizar dichos métodos podríamos llamarlos `ObtenerNombreCurso` y `EstablecerNombreCurso`, en forma respectiva.

Aunque es posible definir métodos como `ObtenerNombreCurso` y `EstablecerNombreCurso`, C# proporciona una solución más elegante. A continuación veremos cómo declarar y utilizar las propiedades.

#### La clase `LibroCalificaciones` con una propiedad

La **declaración de la propiedad** `NombreCurso` de la clase `LibroCalificaciones` se encuentra en las líneas 11-21 de la figura 4.7. La propiedad empieza en la línea 11 con un modificador de acceso (en este caso, `public`), seguido del tipo que representa la propiedad (`string`) y del nombre de la propiedad (`NombreCurso`). Por lo general los nombres de las propiedades empiezan con mayúsculas.

Las propiedades contienen **descriptores de acceso** que se encargan de todos los detalles relacionados con los procesos de devolver y modificar datos. La declaración de una propiedad puede contener un descriptor de acceso `get`, un descriptor de acceso `set` o ambos. El descriptor de acceso `get` (líneas 13-16) permite a un cliente leer el valor de la variable de instancia `private` llamada `nombreCurso`; el descriptor de acceso `set` (líneas 17-20) permite a un cliente modificar el valor de `nombreCurso`.

Después de definir una propiedad, puede utilizarla como una variable en su código. Por ejemplo, puede asignar un valor a una propiedad mediante el uso del operador `=` (asignación). Esto ejecuta el código en el descriptor de acceso `set` de la propiedad, para establecer el valor de la correspondiente variable de instancia. De manera similar, al hacer referencia a la propiedad para utilizar su valor (por ejemplo, para mostrarlo en pantalla) se ejecuta el código en el descriptor de acceso `get` de la propiedad para obtener el valor de la correspondiente variable de instancia. En breve le mostraremos cómo utilizar las propiedades. Por convención, nombramos cada propiedad con el nombre de la variable de instancia que manipula, empezando con letra mayúscula (por ejemplo, `NombreCurso`

es la propiedad que representa a la variable de instancia `nombreCurso`; como C# es sensible a mayúsculas/miúsculas, entonces éstos son identificadores distintos.

### **Descriptores de acceso get y set**

Veamos más de cerca los descriptores de acceso `get` y `set` de la propiedad `NombreCurso` (figura 4.7). El descriptor de acceso `get` (líneas 13-16) empieza con el identificador `get` y está delimitado entre llaves. El cuerpo del descriptor de acceso contiene una **instrucción de retorno**, que consta de la palabra clave `return`, seguida de una expresión. El valor de la expresión se devuelve al código del cliente que está usando la propiedad. En este ejemplo se devuelve el valor de `nombreCurso` cuando se hace referencia a la propiedad `NombreCurso`. Por ejemplo, la siguiente instrucción:

```
string elNombreDelCurso = libroCalificaciones.NombreCurso;
```

en donde `libroCalificaciones` es un objeto de la clase `LibroCalificaciones`, el cual ejecuta el descriptor de acceso `get` de la propiedad `NombreCurso`, que a su vez devuelve el valor de la variable de instancia `nombreCurso`. Después, ese valor se almacena en la variable `elNombreDelCurso`. Observe que la propiedad `NombreCurso` se puede utilizar al igual que una variable de instancia. La notación de la propiedad permite al cliente considerarla como los datos subyacentes. De nuevo, el cliente no puede manipular en forma directa la variable de instancia `nombreCurso`, ya que es `private`.

El descriptor de acceso `set` (líneas 17-20) empieza con el identificador `set` y está delimitado por llaves. Cuando aparece la propiedad `NombreCurso` en una instrucción de asignación, como en

```
libroCalificaciones.NombreCurso = "CS100 Introducción a las computadoras";
```

el texto "CS100 Introducción a las computadoras" se pasa a un parámetro implícito llamado `value` y se ejecuta el descriptor de acceso `set`. Observe que `value` se declara en forma implícita y se inicializa en el descriptor de acceso `set`; sería un error de compilación declarar una variable local llamada `value` en este cuerpo. La línea 19 almacena este valor en la variable de instancia `nombreCurso`. Los descriptores de acceso `set` no devuelven datos cuando completan sus tareas.

Las instrucciones dentro de la propiedad en las líneas 15 y 19 (figura 4.7) tienen acceso a `nombreCurso`, aun y cuando ésta se declaró fuera de la propiedad. Podemos utilizar la variable de instancia `nombreCurso` en los métodos y las propiedades de la clase `LibroCalificaciones`, ya que `nombreCurso` es una variable de instancia de la clase. El orden en el que se declaran los métodos y las propiedades en una clase no determina cuándo se van a llamar en tiempo de ejecución, por lo que podemos declarar el método `MostrarMensaje` (que utiliza la propiedad `NombreCurso`) antes de declarar la propiedad `NombreCurso`. Dentro de la propiedad en sí, los descriptores de acceso `get` y `set` pueden aparecer en cualquier orden, y es posible omitir cualquiera de ellos. En el capítulo 9 veremos cómo omitir un descriptor de acceso `get` o `set` para crear las denominadas propiedades de "sólo lectura" y "sólo escritura", en forma respectiva.

### **Uso de la propiedad `NombreCurso` en el método `MostrarMensaje`**

El método `MostrarMensaje` (líneas 24-30 de la figura 4.7) no recibe parámetros. Las líneas 28-29 imprimen en pantalla un mensaje de bienvenida, que incluye el valor de la variable de instancia `nombreCurso`. No hacemos referencia a `nombreCurso` en forma directa, sino que accedemos a la propiedad `NombreCurso` (línea 29), que ejecuta su descriptor de acceso `get` y así devuelve el valor de `nombreCurso`.

### **La clase `PruebaLibroCalificaciones` para demostrar la clase `LibroCalificaciones`**

La clase `PruebaLibroCalificaciones` (figura 4.8) crea un objeto `LibroCalificaciones` y demuestra el uso de la propiedad `NombreCurso`. La línea 11 crea un objeto `LibroCalificaciones` y lo asigna a la variable local `miLibroCalificaciones`, de tipo `LibroCalificaciones`. Las líneas 14-15 muestran el nombre inicial del curso mediante el uso de la propiedad `NombreCurso` del objeto; con esto se ejecuta el descriptor de acceso `get` de la propiedad, el cual devuelve el valor `nombreCurso`.

Observe que la primera línea de la salida muestra un nombre vacío (marcado con ' '). A diferencia de las variables locales que no se inicializan en forma automática, cada campo tiene un **valor inicial predeterminado**: un valor que proporciona C# cuando usted no especifica el valor inicial. Por ende, no se requiere inicializar los

```

1 // Fig. 4.8: PruebaLibroCalificaciones.cs
2 // Creación y manipulación de un objeto LibroCalificaciones.
3 using System;
4
5 public class PruebaLibroCalificaciones
6 {
7     // El método Main comienza la ejecución del programa
8     public static void Main( string[] args )
9     {
10         // crea un objeto LibroCalificaciones y lo asigna a miLibroCalificaciones
11         LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones();
12
13         // muestra el valor inicial de NombreCurso
14         Console.WriteLine( "El nombre inicial del curso es: '{0}'\n",
15             miLibroCalificaciones.NombreCurso );
16
17         // pide y lee el nombre del curso
18         Console.WriteLine( "Por favor escriba el nombre del curso:" );
19         string elNombre = Console.ReadLine(); // lee una línea de texto
20         miLibroCalificaciones.NombreCurso = elNombre; // establece el nombre usando una
21         // propiedad
22         Console.WriteLine(); // imprime en pantalla una línea en blanco
23
24         // muestra el mensaje de bienvenida después de especificar el nombre del curso
25         miLibroCalificaciones.MostrarMensaje();
26     } // fin de Main
27 } // fin de la clase PruebaLibroCalificaciones

```

```

El nombre inicial del curso es: ''

Por favor escriba el nombre del curso:
CS101 Introducción a la programación en C#

¡Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C!

```

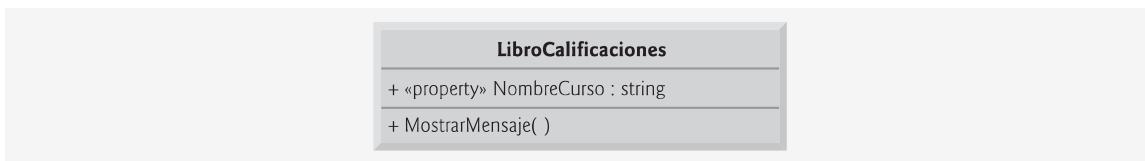
**Figura 4.8 |** Creación y manipulación de un objeto LibroCalificaciones.

campos en forma explícita antes de utilizarlos en una aplicación; a menos que deban inicializarse con valores distintos de los predeterminados. El valor predeterminado para una variable de instancia de tipo `string` (como `NombreCurso`) es `null`. Cuando se muestra en pantalla una variable `string` que contiene el valor `null`, no se muestra ningún texto. En la sección 4.8 hablaremos sobre el significado de `null`.

La línea 18 pide al usuario que escriba el nombre para el curso. La variable `string` local `elNombre` (declarada en la línea 19) se inicializa con el nombre del curso que escribió el usuario, el cual se devuelve mediante la llamada a `ReadLine`. La línea 20 asigna `elNombre` a la propiedad `NombreCurso` del objeto `miLibroCalificaciones`. Cuando se asigna un valor a `NombreCurso`, el valor especificado (en este caso, `elNombre`) se asigna al parámetro implícito `value` del descriptor de acceso `set` de `NombreCurso` (líneas 17-20, figura 4.7). Después el parámetro `value` se asigna mediante el descriptor de acceso `set` a la variable de instancia `NombreCurso` (línea 19 de la figura 4.7). La línea 21 (figura 4.8) muestra en pantalla una línea en blanco y después la línea 24 llama al método `MostrarMensaje` de `miLibroCalificaciones` para mostrar el mensaje de bienvenida que contiene el nombre del curso.

## 4.6 Diagrama de clases de UML con una propiedad

La figura 4.9 contiene un diagrama de clases de UML actualizado para la versión de `LibroCalificaciones` de la figura 4.7. En UML las propiedades se modelan como atributos; la propiedad (en este caso, `NombreCurso`) se lista como un atributo público, como lo indica el signo más (+), y se le antepone la palabra “property” encerrada entre



**Figura 4.9** | Diagrama de clases de UML, en el que se indica que la clase `LibroCalificaciones` tiene una propiedad pública `NombreCurso` de tipo `string` y un método público.

los signos « y ». El uso de palabras descriptivas entre los signos « y » (lo que se conoce como **estereotipos** en UML) ayuda a distinguir las propiedades de otros atributos y operaciones. Para indicar el tipo de propiedad, UML coloca un signo de dos puntos y un tipo después del nombre de la propiedad. Los descriptores de acceso `get` y `set` de la propiedad están implícitos, por lo que no se listan en el diagrama de UML. La clase `LibroCalificaciones` también contiene un método `public` llamado `MostrarMensaje`, por lo que el diagrama de clases lista esta operación en el tercer compartimiento. Recuerde que el signo más (+) es el símbolo de visibilidad pública.

En la sección anterior aprendió a declarar una propiedad en código de C#. Vio que por lo general a una propiedad se le da el mismo nombre que la variable de instancia que manipula, pero con la primera letra en mayúscula (por ejemplo, la propiedad `NombreCurso` manipula a la variable de instancia `nombreCurso`). Un diagrama de clases le ayuda a diseñar una clase, por lo que no se requiere mostrar todos los detalles de implementación de la clase. Debido a que una variable de instancia manipulada por una propiedad es en realidad un detalle de implementación de esa propiedad, nuestro diagrama de clases no muestra la variable de instancia `nombreCurso`. Un programador que implemente la clase `LibroCalificaciones` con base en este diagrama de clases crearía la variable de instancia `nombreCurso` como parte del proceso de implementación (como hicimos en la figura 4.7).

En ocasiones, tal vez sea necesario modelar las variables de instancia `private` de una clase que no sean propiedades. Al igual que las propiedades, las variables de instancia son atributos de una clase y se modelan en el compartimiento de en medio de un diagrama de clases. Para representar las variables de instancia como atributos, UML lista el nombre del atributo, seguido de un signo de dos puntos y del tipo del atributo. Para indicar que un atributo es `private`, un diagrama de clases listaría el **símbolo de visibilidad privada** [un signo menos (-)] antes del nombre del atributo. Por ejemplo, la variable de instancia `nombreCurso` en la figura 4.7 se modelaría como “`- nombreCurso : string`” para indicar que es un atributo privado de tipo `string`.

## 4.7 Ingeniería de software con propiedades y los descriptores de acceso set y get

El uso de las propiedades como se describió antes en este capítulo parece violar la noción de los datos `private`. Aunque proporcionar una propiedad con descriptores de acceso `get` y `set` podría parecer lo mismo que hacer que su correspondiente variable de instancia sea `public`, en realidad no es así. Cualquier propiedad o método en el programa puede leer o escribir en una variable de instancia `public`. Si una variable de instancia es `private`, el código cliente puede acceder a esa variable de instancia sólo en forma indirecta, a través de las propiedades o métodos `private` de la clase. Esto permite a la clase controlar la forma en la que se establecen o se devuelven los datos. Por ejemplo, los descriptores de acceso `get` y `set` pueden traducir entre el formato de los datos que utiliza el cliente y el formato almacenado en la variable de instancia `private`.

Considere una clase llamada `Reloj`, que representa la hora del día como una variable de instancia `private int` llamada `tiempo`, que contiene el número de segundos transcurridos desde medianoche. Suponga que la clase cuenta con una propiedad `Tiempo` de tipo `string` para manipular esta variable de instancia. Aunque por lo general los descriptores de acceso `get` devuelven los datos en la misma forma exacta en la que están almacenados en un objeto, no necesitan exponer los datos en este formato “puro”. Cuando un cliente hace referencia a la propiedad `Tiempo` de un objeto `Reloj`, el descriptor de acceso `get` de la propiedad puede utilizar la variable de instancia `tiempo` para determinar el número de horas, minutos y segundos transcurridos desde medianoche, para después devolver la hora como un objeto `string` de la forma “`HH:MM:SS`”. De manera similar, suponga que se asigna un objeto `string` de la forma “`HH:MM:SS`” a la propiedad `Tiempo` de un objeto `Reloj`. Mediante el uso de las capacidades de `string` que se presentan en el capítulo 16 y el método `Convert.ToInt32` presentado en la sección 3.6, el descriptor de acceso `set` de la propiedad `Tiempo` podría convertir este objeto `string` en un número `int`.

de segundos transcurridos a partir de medianoche, y almacenar el resultado en la variable de instancia `private` `tiempo` del objeto `Reloj`. El descriptor de acceso `set` de la propiedad `Tiempo` también puede proporcionar capacidades de **validación de datos**, para escudriñar los intentos de modificar el valor de la variable de instancia y asegurar que el valor que reciba represente una hora válida (por ejemplo: "12:30:45" es válida, pero "42:85:70" no). En la sección 4.10 demostraremos el uso de la validación de datos. Así, aunque los descriptores de acceso de una propiedad permiten a los clientes manipular los datos `private`, controlan con cuidado esas manipulaciones y los datos `private` del objeto permanecen encapsulados (es decir, ocultos) en forma segura dentro del objeto. Esto no es posible con las variables de instancia `public`, que los clientes pueden establecer con facilidad y asignarles valores inválidos.

También es conveniente que los propios métodos de la clase utilicen las propiedades de la misma para manipular sus variables de instancia `private`, aun y cuando los métodos pueden acceder en forma directa a las variables de instancia `private`. Al acceder a una variable de instancia a través de los descriptores de acceso de una propiedad, como en el cuerpo del método `MostrarMensaje` (figura 4.7, líneas 28-29), se crea una clase más robusta que es más fácil de mantener y tiene menos probabilidad de fallar. Si decidimos modificar la representación de la variable de instancia `nombreCurso` de alguna forma, la declaración del método `MostrarMensaje` no requiere modificación; sólo tendrán que cambiar los cuerpos de los descriptores de acceso `get` y `set` de la propiedad `NombreCurso` que manipulan en forma directa a la variable de instancia. Por ejemplo, suponga que deseamos representar el nombre del curso como dos variables de instancia separadas: `numeroCurso` (por ejemplo, "CS101") y `tituloCurso` (por ejemplo, "Introducción a la programación en C#"). El método `MostrarMensaje` puede seguir utilizando el descriptor de acceso `get` de la propiedad `NombreCurso` para obtener el nombre completo del curso y mostrarlo como parte del mensaje de bienvenida. En este caso, el descriptor de acceso `get` tendría que crear y devolver un objeto `string` con el valor de `numeroCurso`, seguido del valor de `tituloCurso`. El método `MostrarMensaje` seguiría mostrando en pantalla el título completo del curso "CS101 Introducción a la programación en C#", debido a que no se ve afectado por la modificación de las variables de instancia de la clase.



### Observación de ingeniería de software 4.3

*Al acceder a los datos `private` a través de los descriptores de acceso `set` y `get` no sólo se protegen las variables de instancia de recibir valores inválidos, sino que también se oculta la representación interna de las variables de instancia de los clientes de esa clase. Por lo tanto, si cambia la representación de los datos (a menudo para reducir la cantidad de almacenamiento requerido o para mejorar el rendimiento), sólo necesitan cambiar las implementaciones de las propiedades; las implementaciones de los clientes no necesitan modificarse siempre y cuando se preserven los servicios que proporcionan las propiedades.*

## 4.8 Comparación entre tipos por valor y tipos por referencia

En C#, los tipos se dividen en dos categorías: **por valor** y **por referencia**. Todos los tipos simples de C# son por valor; una variable de este tipo (como `int`) tan sólo contiene un valor de ese tipo. Por ejemplo, la figura 4.10 muestra una variable `int` llamada `conteo`, la cual contiene el valor 7.

En contraste, una variable de un tipo por referencia (también conocida como **referencia**) contiene la dirección de una ubicación en memoria, en donde se almacenan los datos a los que hace referencia. Se dice que dicha variable **hace referencia a un objeto** en el programa. La línea 11 de la figura 4.8 crea un objeto `LibroCalificaciones`, lo coloca en memoria y almacena la dirección de memoria de ese objeto en la variable de referencia `miLibroCalificaciones` de tipo `LibroCalificaciones`, como se muestra en la figura 4.11. Observe que el objeto `LibroCalificaciones` se muestra con su variable de instancia `nombreCurso`.

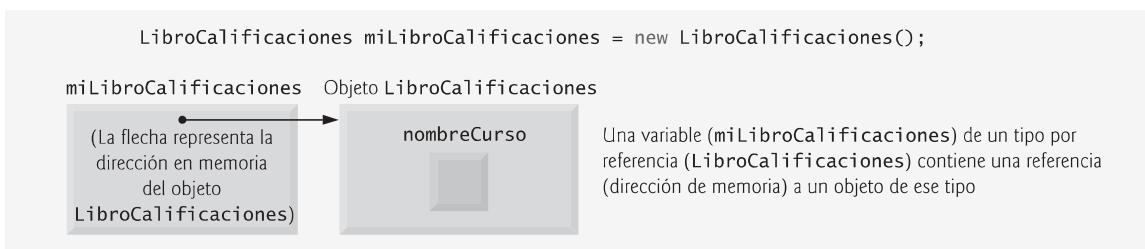
```
int conteo = 7;
```

conteo

7

Una variable (conteo) de un tipo por valor (`int`)  
contiene un valor (7) de ese tipo

Figura 4.10 | Variable de tipo por valor.



**Figura 4.11** | Variable de tipo por referencia.

Las variables de instancia de tipo por referencia (como **miLibroCalificaciones** en la figura 4.11) se inicializan de manera predeterminada con el valor **null**. **string** es un tipo por referencia; por esta razón, la variable **string nombreCurso** se muestra en la figura 4.11 con un cuadro vacío, que representa a la variable con valor **null** en memoria.

Un cliente de un objeto debe utilizar una referencia a ese objeto para **invocar** (es decir, llamar) a los métodos del objeto y acceder a sus propiedades. En la figura 4.8, las instrucciones en **Main** utilizan la variable **miLibroCalificaciones**, que contiene la referencia al objeto **LibroCalificaciones**, para enviar mensajes al objeto **LibroCalificaciones**. Estos mensajes son llamadas a métodos (como **MostrarMensaje**) o referencias a propiedades (como **nombreCurso**), las cuales permiten al programa interactuar con objetos tipo **LibroCalificaciones**. Por ejemplo, la instrucción (en la línea 20 de la figura 4.8)

```
miLibroCalificaciones.NombreCurso = elNombre; // establece el nombre usando una propiedad
```

utiliza la referencia **miLibroCalificaciones** para establecer el nombre del curso mediante la asignación de un valor a la propiedad **NombreCurso**. Esto envía un mensaje al objeto **LibroCalificaciones** para que invoque al descriptor de acceso **set** de la propiedad **NombreCurso**. El mensaje incluye como argumento el valor "**CS101 Introducción a la programación en C#**" que requiere el descriptor de acceso **set** de **NombreCurso** para realizar su tarea. El descriptor de acceso **set** utiliza esta información para establecer la variable de instancia **nombreCurso**. En la sección 7.14 hablaremos con detalle acerca de los tipos por valor y los tipos por referencia.



#### Observación de ingeniería de software 4.4

*El tipo declarado de una variable (por ejemplo, **int**, **double** o **LibroCalificaciones**) indica si la variable es de un tipo por valor o por referencia. Si el tipo de una variable no es uno de los trece tipos simples, o un tipo enum o struct (que veremos en la sección 7.10 y en el capítulo 16, en forma respectiva), entonces es un tipo por referencia. Por ejemplo, **Cuenta cuenta1** indica que **cuenta1** es una variable que puede hacer referencia a un objeto **Cuenta**.*

## 4.9 Inicialización de objetos con constructores

Como mencionamos en la sección 4.5, cuando se crea un objeto de la clase **LibroCalificaciones** (figura 4.7), su variable de instancia **nombreCurso** se inicializa con **null** de manera predeterminada. ¿Qué pasa si usted desea proporcionar el nombre de un curso cuando crea un objeto **LibroCalificaciones**? Cada clase que usted declare puede proporcionar un **constructor**, que puede utilizarse para inicializar un objeto de una clase al momento de crear ese objeto. De hecho, C# requiere una llamada al constructor para cada objeto que se crea. El operador **new** llama al constructor de la clase para realizar la inicialización. La llamada al constructor se indica mediante el nombre de la clase, seguido de paréntesis. Por ejemplo, la línea 11 de la figura 4.8 primero utiliza **new** para crear un objeto **LibroCalificaciones**. Los paréntesis vacíos después de "**new LibroCalificaciones**" indican una llamada sin argumentos al constructor de la clase. De manera predeterminada, el compilador proporciona un **constructor predeterminado** sin parámetros, en cualquier clase que no incluya un constructor en forma explícita, por lo que *toda* clase tiene un constructor.

Cuando usted declara una clase, puede proporcionar su propio constructor para especificar una inicialización personalizada para los objetos de su clase. Por ejemplo, tal vez quiera especificar el nombre de un curso para un objeto **LibroCalificaciones** cuando se crea este objeto, como en

```

LibroCalificaciones miLibroCalificaciones =
    new LibroCalificaciones( "CS101 Introducción a la programación en C#" );

```

En este caso, el argumento "CS101 Introducción a la programación en C#" se pasa al constructor del objeto `LibroCalificaciones` y se utiliza para inicializar a `nombreCurso`. Cada vez que usted crea un objeto `LibroCalificaciones` distinto, puede proporcionar un nombre distinto para el curso. La instrucción anterior requiere que la clase proporcione un constructor con un parámetro `string`. La figura 4.12 contiene una clase `LibroCalificaciones` modificada con dicho constructor.

Las líneas 10-13 declaran el constructor para la clase `LibroCalificaciones`. Un constructor debe tener el mismo nombre que su clase. Al igual que un método, un constructor especifica en su lista de parámetros los datos que requiere para realizar su tarea. A diferencia de un método, un constructor no especifica un tipo de valor de retorno. Cuando usted crea un nuevo objeto (con `new`), coloca estos datos en los paréntesis que van después del nombre de la clase. La línea 10 indica que el constructor de la clase `LibroCalificaciones` tiene un parámetro llamado `nombre`, de tipo `string`. En la línea 12 del cuerpo del constructor, el `nombre` que se pasa al constructor se asigna a la variable de instancia `nombreCurso` mediante el descriptor de acceso `set` de la propiedad `NombreCurso`.

```

1 // Fig. 4.12: LibroCalificaciones.cs
2 // La clase LibroCalificaciones con un constructor para inicializar el nombre del curso.
3 using System;
4
5 public class LibroCalificaciones
6 {
7     private string nombreCurso; // nombre del curso para este LibroCalificaciones
8
9     // el constructor inicializa nombreCurso con el objeto string suministrado como
10    // argumento
11    public LibroCalificaciones( string nombre )
12    {
13        NombreCurso = nombre; // inicializa nombreCurso usando la propiedad
14    } // fin del constructor
15
16    // propiedad para obtener (get) y establecer (set) el nombre del curso
17    public string NombreCurso
18    {
19        get
20        {
21            return nombreCurso;
22        } // fin de get
23        set
24        {
25            nombreCurso = value;
26        } // fin de set
27    } // fin de la propiedad NombreCurso
28
29    // muestra un mensaje de bienvenida para el usuario del LibroCalificaciones
30    public void MostrarMensaje()
31    {
32        // usa la propiedad NombreCurso para obtener el
33        // nombre del curso que representa este LibroCalificaciones
34        Console.WriteLine( "Bienvenido al libro de calificaciones para\n{0}!",
35            NombreCurso );
36    } // fin del método MostrarMensaje
37 } // fin de la clase LibroCalificaciones

```

**Figura 4.12** | La clase `LibroCalificaciones` con un constructor para inicializar el nombre del curso.

La figura 4.13 demuestra la inicialización de objetos `LibroCalificaciones` mediante el uso de este constructor. Las líneas 12-13 crean e inicializan un objeto `LibroCalificaciones`. El constructor de la clase `LibroCalificaciones` se llama con el argumento "CS101 Introducción a la programación en C#" para inicializar el nombre del curso. La expresión de creación de objeto a la derecha del signo = en las líneas 12-13 devuelve una referencia al nuevo objeto, el cual se asigna a la variable `libroCalificaciones1`. Las líneas 14-15 repiten este proceso para otro objeto `LibroCalificaciones`, pero esta vez pasan el argumento "CS102 Estructuras de datos en C#" para inicializar el nombre del curso para `libroCalificaciones2`. Las líneas 18-21 utilizan la propiedad `NombreCurso` de cada objeto para obtener los nombres de los cursos y mostrar que sin duda se inicializaron en el momento en el que se crearon. En la introducción a la sección 4.5, usted aprendió que cada instancia (es decir, objeto) de una clase contiene su propia copia de las variables de instancia de la clase. La salida confirma que cada `LibroCalificaciones` mantiene su propia copia de la variable `nombreCurso`.

Al igual que los métodos, los constructores también pueden recibir argumentos. No obstante, una importante diferencia entre los constructores y los métodos es que los primeros no pueden devolver valores; de hecho, no pueden especificar un tipo de valor de retorno (ni siquiera `void`). Por lo general, los constructores se declaran como `public`. Si una clase no incluye un constructor, las variables de instancia de esa clase se inicializan con sus valores predeterminados. Si usted declara uno o más constructores para una clase, C# no creará un constructor predeterminado para esa clase.



### Tip de prevención de errores 4.1

*A menos que sea aceptable la inicialización predeterminada de las variables de instancia de su clase, deberá proporcionar un constructor para asegurarse que las variables de instancia de su clase se inicialicen en forma apropiada con valores significativos, cuando se cree cada nuevo objeto de su clase.*

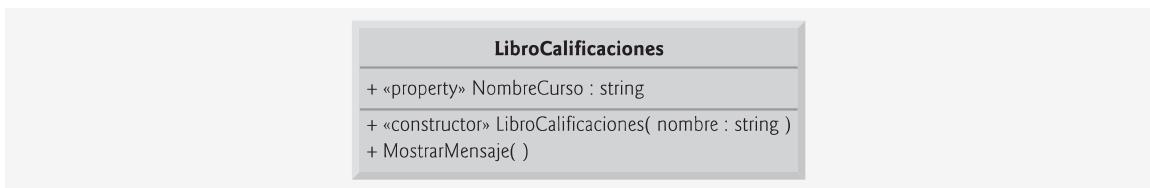
```

1 // Fig. 4.13: PruebaLibroCalificaciones.cs
2 // El constructor LibroCalificaciones se utiliza para especificar el nombre del
3 // curso cada vez que se crea un objeto LibroCalificaciones.
4 using System;
5
6 public class PruebaLibroCalificaciones
7 {
8     // El método Main comienza la ejecución del programa
9     public static void Main( string[] args )
10    {
11        // crea el objeto LibroCalificaciones
12        LibroCalificaciones libroCalificaciones1 = new LibroCalificaciones( // invoca al
13            "CS101 Introducción a la programación en C#" );
14        LibroCalificaciones libroCalificaciones2 = new LibroCalificaciones( // invoca al
15            "CS102 Estructuras de datos en C#" );
16
17        // muestra el valor inicial de nombreCurso para cada LibroCalificaciones
18        Console.WriteLine( "El nombre del curso de libroCalificaciones1 es: {0}",
19            libroCalificaciones1.NombreCurso );
20        Console.WriteLine( "El nombre del curso de libroCalificaciones2 es: {0}",
21            libroCalificaciones2.NombreCurso );
22    } // fin de Main
23 } // fin de la clase PruebaLibroCalificaciones

```

El nombre del curso de `libroCalificaciones1` es: CS101 Introducción a la programación en C#  
 El nombre del curso de `libroCalificaciones2` es: CS102 Estructuras de datos en C#

**Figura 4.13** | El constructor de `LibroCalificaciones` se utiliza para especificar el nombre del curso cada vez que se crea cada un objeto `LibroCalificaciones`.



**Figura 4.14** | Diagrama de clases de UML donde se indica que la clase *LibroCalificaciones* tiene un constructor con el parámetro *Name* del tipo *string*.

#### **Agregar el constructor al diagrama de clases de UML de la clase *LibroCalificaciones***

El diagrama de clases de UML de la figura 4.14 modela la clase *LibroCalificaciones* de la figura 4.12, que tiene un constructor con un parámetro llamado *nombreCurso*, de tipo *string*. Al igual que las operaciones, el diagrama de clases UML modela a los constructores en el tercer compartimiento de una clase. Para diferenciar a un constructor de las operaciones de una clase, UML coloca la palabra “constructor” entre los signos « y » antes del nombre del constructor. Es costumbre listar los constructores antes de otras operaciones en el tercer compartimiento.

## **4.10 Los números de punto flotante y el tipo decimal**

En nuestra siguiente aplicación dejaremos por un momento nuestro caso de estudio con la clase *LibroCalificaciones* para declarar una clase llamada *Cuenta*, que mantiene el saldo de una cuenta bancaria. La mayoría de los saldos de las cuentas no son números enteros (por ejemplo, 0, -22 y 1024). Por esta razón, la clase *Cuenta* representa el saldo de las cuentas como un número real (es decir, un número con un punto decimal, como 7.33, 0.0975 o 1000.12345). C# proporciona tres tipos simples para almacenar números reales en memoria: *float*, *double* y *decimal*. A los tipos *float* y *double* se les llama tipos de *punto flotante*. La principal diferencia entre ellos y *decimal* es que las variables tipo *decimal* almacenan un rango limitado de números reales con precisión, mientras que las variables de punto flotante sólo almacenan aproximaciones de números reales, pero a través de un rango mucho mayor de valores. Además, las variables *double* pueden almacenar números con mayor magnitud y detalle (es decir, más dígitos a la derecha del punto decimal; también se le conoce como la *precisión* del número) que las variables *float*. Una aplicación clave del tipo *decimal* es para representar cantidades monetarias.

#### **Precisión de los números reales y requerimientos de memoria**

Las variables de tipo *float* representan *números de punto flotante de precisión simple* y tienen siete dígitos significativos. Las variables de tipo *double* representan *números de punto flotante de precisión doble*. Estos requieren el doble de memoria que las variables *float* y proporcionan de 15 a 16 dígitos significativos; aproximadamente el doble de precisión de las variables *float*. Lo que es más, las variables de tipo *decimal* requieren hasta el doble de memoria que las variables *double* y proporcionan de 28 a 29 dígitos significativos. Para el rango de valores requeridos por la mayoría de las aplicaciones debe bastar con las variables de tipo *float* para las aproximaciones, pero podemos utilizar variables tipo *double* o *decimal* para estar seguros. En algunas aplicaciones, incluso hasta las variables de tipo *double* y *decimal* serán inadecuadas; dichas aplicaciones se encuentran más allá del alcance de este libro.

La mayoría de los programadores representa números de punto flotante con el tipo *double*. De hecho, C# trata a todos los números reales que uno escribe en el código fuente de una aplicación (como 7.33 y 0.0975) como valores *double* de manera predeterminada. Dichos valores en el código fuente se conocen como *literales de punto flotante*. Para escribir una *literal decimal*, debe escribir la letra “M” o “m” al final de un número real (por ejemplo, 7.33M es una literal *decimal* en vez de *double*). Las literales enteras se convierten de manera implícita en literales tipo *float*, *double* o *decimal* cuando se asignan a una variable de uno de estos tipos. En el apéndice L, Los tipos simples, podrá consultar los rangos de los valores para los tipos *float*, *double*, *decimal* y todos los demás tipos simples.

Aunque los números de punto flotante no son siempre 100% precisos, tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal “normal” de 36.8, no necesitamos una precisión con un número extenso de dígitos. Cuando leemos la temperatura en un termómetro como 36.8, en realidad podría ser 36.7999473210643. Si consideramos a este número simplemente como 36.8 está bien para la mayoría de las

aplicaciones en las que se trabaja con las temperaturas corporales. Debido a la naturaleza imprecisa de los números de punto flotante, se prefiere el tipo `decimal` a los tipos de punto flotante siempre que los cálculos necesitan ser exactos, como en los cálculos monetarios. En los casos en donde es suficiente con las aproximaciones, se prefiere el tipo `double` al tipo `float` ya que las variables `double` pueden representar los números de punto flotante con más precisión. Por esta razón, utilizaremos el tipo `decimal` a lo largo de este libro para tratar con cantidades monetarias y el tipo `double` para los demás números reales.

Los números reales también surgen como resultado de la división. Por ejemplo, en la aritmética convencional, cuando dividimos 10 entre 3 el resultado es 3.333333..., y la secuencia de números 3 se repite en forma indefinida. La computadora asigna sólo una cantidad fija de espacio para almacenar un valor de este tipo, por lo que es evidente que el valor de punto flotante almacenado sólo puede ser una aproximación.



### Error común de programación 4.3

*El uso de números de punto flotante de una manera en la que se asuma que se representan con precisión puede producir errores lógicos.*

#### ***La clase Cuenta con una variable de instancia de tipo decimal***

Nuestra siguiente aplicación (figuras 4.15-4.16) contiene una clase muy simplificada llamada `Cuenta` (figura 4.15), que mantiene el saldo de una cuenta bancaria. Un banco ordinario da servicio a muchas cuentas, cada una con su propio balance, por lo que la línea 7 declara una variable de instancia llamada `saldo` de tipo `decimal`. La variable `saldo` es una variable de instancia, ya que está declarada en el cuerpo de la clase (líneas 6-36) pero fuera del método de la clase y de las declaraciones de las propiedades (líneas 10-13, 16-19 y 22-35). Cada instancia (es decir, objeto) de la clase `Cuenta` contiene su propia copia de `saldo`.

La clase `Cuenta` contiene un constructor, un método y una propiedad. Como es común que alguien abra una cuenta para depositar dinero de inmediato, el constructor (líneas 10-13) recibe un parámetro llamado `saldoInicial` de tipo `decimal`, que representa el saldo inicial de la cuenta. La línea 12 asigna `saldoInicial` a la propiedad `Saldo`, con lo cual invoca el descriptor de acceso `set` de `Saldo` para inicializar la variable de instancia `saldo`.

El método `Acredita` (líneas 16-19) no devuelve ningún dato cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. El método recibe un parámetro llamado `monto`: un valor `decimal` que se suma a la propiedad `Saldo`. La línea 18 utiliza los descriptores de acceso `get` y `set` de `Saldo`. La expresión `Saldo + monto` invoca al descriptor de acceso `get` de `Saldo` para obtener el valor actual de la variable de instancia `saldo` y después le suma la variable `monto`. Posteriormente asignamos el resultado a la variable de instancia `saldo` mediante la invocación del descriptor de acceso `set` de la propiedad `Saldo` (con lo cual se sustituye el valor anterior de `saldo`).

La propiedad `Saldo` (líneas 22-35) cuenta con un descriptor de acceso `get`, que permite a los clientes de la clase (es decir, otras clases que utilicen esta clase) obtener el valor del `saldo` del objeto de una `Cuenta` específica. La propiedad tiene el tipo `decimal` (línea 22). `Saldo` también cuenta con un descriptor de acceso `set` mejorado.

En la sección 4.5 presentamos las propiedades cuyos descriptores de acceso `set` permiten a los clientes de la clase modificar el valor de una variable de instancia `private`. En la figura 4.7, la clase `LibroCalificaciones` define el descriptor de acceso `set` de la propiedad `NombreCurso` para asignar el valor que recibe en su parámetro `value` a la variable de instancia `nombreCurso` (línea 19). Esta propiedad `NombreCurso` no asegura que `NombreCurso` sólo contenga datos válidos.

La aplicación de las figuras 4.15 a la 4.16 mejora el descriptor de acceso `set` de la propiedad `Saldo` de la clase `Cuenta` para realizar esta validación (lo que también se conoce como **comprobación de validez**). La línea 32 (figura 4.15) asegura que el valor no sea negativo. Si el valor es mayor o igual a 0, el monto almacenado en `value` se asigna a la variable de instancia `saldo` en la línea 33. De no ser así, `saldo` permanece sin cambios.

#### ***La clase PruebaCuenta que utiliza a la clase Cuenta***

La clase `PruebaCuenta` (figura 4.16) crea dos objetos `Cuenta` (líneas 10-11) y los inicializa en forma respectiva con `50.00M` y `-7.53M` (las literales decimales que representan los números reales `50.00` y `-7.53`). Observe que el constructor de `Cuenta` (líneas 10-13 de la figura 4.15) hace referencia a la propiedad `Saldo` para inicializar

```

1 // Fig. 4.15: Cuenta.cs
2 // La clase Cuenta con un constructor para
3 // inicializar la variable de instancia saldo.
4
5 public class Cuenta
6 {
7     private decimal saldo; // variable de instancia que almacena el saldo
8
9     // constructor
10    public Cuenta( decimal saldoInicial )
11    {
12        Saldo = saldoInicial; // establece el saldo usando la propiedad
13    } // fin del constructor de Cuenta
14
15    // acredita (suma) un monto a la cuenta
16    public void Acredita( decimal monto )
17    {
18        Saldo = Saldo + monto; // suma monto al saldo
19    } // fin del método Acredita
20
21    // una propiedad para obtener (get) y establecer (set) el saldo de una cuenta
22    public decimal Saldo
23    {
24        get
25        {
26            return saldo;
27        } // end get
28        set
29        {
30            // valida que el valor sea mayor o igual a 0;
31            // si no lo es, el saldo permanece sin cambios
32            if ( value >= 0 )
33                saldo = value;
34        } // fin de set
35    } // fin de la propiedad Saldo
36 } // fin de la clase Cuenta

```

**Figura 4.15** | La clase Cuenta con un constructor para inicializar la variable de instancia saldo.

saldo. En ejemplos anteriores, el beneficio de hacer referencia a la propiedad en el constructor no era evidente. Sin embargo, ahora el constructor aprovecha la validación que proporciona el descriptor de acceso set de la propiedad Saldo. Y sólo asigna un valor a Saldo, en vez de duplicar el código de validación del descriptor de acceso set. Cuando la línea 11 de la figura 4.16 pasa un saldo inicial de -7.53 al constructor de Cuenta, el constructor pasa este valor al descriptor de acceso set de la propiedad Saldo, en donde ocurre la inicialización en sí. Este valor es menor que 0, por lo que el descriptor de acceso set no modifica a saldo y deja a esta variable de instancia con su valor predeterminado de 0.

Las líneas de la 14 a la 17 en la figura 4.16 imprimen en pantalla el saldo en cada Cuenta mediante el uso de la propiedad Saldo de Cuenta. Cuando se utiliza Saldo para cuenta1 (línea 15), el descriptor de acceso get devuelve el valor del saldo de cuenta1 en la línea 26 de la figura 4.15 y la instrucción Console.WriteLine lo muestra en pantalla (figura 4.16, líneas 14-15). De manera similar, cuando se llama a la propiedad Saldo para cuenta2 desde la línea 17, se devuelve el valor del saldo de cuenta2 desde la línea 26 de la figura 4.15 y la instrucción Console.WriteLine la imprime en pantalla (figura 4.16, líneas 16-17). Observe que el saldo de cuenta2 es 0, ya que el constructor se aseguró que la cuenta no pudiera comenzar con un saldo negativo. El valor se imprime en pantalla mediante WriteLine con el elemento de formato {0:C}, el cual da formato al saldo de la cuenta como una cantidad monetaria. El signo : después del 0 indica que el siguiente carácter representa un **especificador de formato**, y el especificador de formato C antes del signo : especifica una cantidad monetaria (C es para moneda).

```

1 // Fig. 4.16: PruebaCuenta.cs
2 // Creación y manipulación de un objeto Cuenta.
3 using System;
4
5 public class PruebaCuenta
6 {
7     // El método Main comienza la ejecución de la aplicación de C#
8     public static void Main( string[] args )
9     {
10         Cuenta cuenta1 = new Cuenta( 50.00M ); // crea el objeto Cuenta
11         Cuenta cuenta2 = new Cuenta( -7.53M ); // crea el objeto Cuenta
12
13         // muestra el saldo inicial de cada objeto usando una propiedad
14         Console.WriteLine( "Saldo de cuenta1: {0:C}",
15             cuenta1.Saldo ); // muestra la propiedad Saldo
16         Console.WriteLine( "Saldo de cuenta2: {0:C}\n",
17             cuenta2.Saldo ); // muestra la propiedad Saldo
18
19         decimal montoDeposito; // deposita la cantidad que se leyó del usuario
20
21         // pide y obtiene la entrada del usuario
22         Console.Write( "Escriba el monto a depositar para la cuenta1: " );
23         montoDeposito = Convert.ToDecimal( Console.ReadLine() );
24         Console.WriteLine( "se sumó {0:C} al saldo de cuenta1\n",
25             montoDeposito );
26         cuenta1.Acredita( montoDeposito ); // se suma al saldo de cuenta1
27
28         // muestra los saldos
29         Console.WriteLine( "Saldo de cuenta1: {0:C}",
30             cuenta1.Saldo );
31         Console.WriteLine( "Saldo de cuenta2: {0:C}\n",
32             cuenta2.Saldo );
33
34         // pide y obtiene la entrada del usuario
35         Console.Write( "Escriba la cantidad a depositar para la cuenta2: " );
36         montoDeposito = Convert.ToDecimal( Console.ReadLine() );
37         Console.WriteLine( "se sumó {0:C} al saldo de cuenta2\n",
38             montoDeposito );
39         cuenta2.Acredita( montoDeposito ); // se suma al saldo de cuenta2
40
41         // muestra los saldos
42         Console.WriteLine( "Saldo de cuenta1: {0:C}", cuenta1.Saldo );
43         Console.WriteLine( "Saldo de cuenta2: {0:C}", cuenta2.Saldo );
44     } // fin de Main
45 } // fin de la clase PruebaCuenta

```

```

Saldo de cuenta1: $50.00
Saldo de cuenta2: $0.00

Escriba el monto a depositar para la cuenta1: 49.99
se sumó $49.99 al saldo de cuenta1

Saldo de cuenta1: $99.99
Saldo de cuenta2: $0.00

Escriba la cantidad a depositar para la cuenta2: 123.21
se sumó $123.21 al saldo de cuenta2

Saldo de cuenta1: $99.99
Saldo de cuenta2: $123.21

```

**Figura 4.16** | Creación y manipulación de un objeto Cuenta.

Especificador de formato	Descripción
C o c	Da formato al objeto string como moneda. Coloca antes del número un símbolo de moneda apropiado (\$) en Estados Unidos y en México). Separa los dígitos con un carácter separador apropiado (coma en Estados Unidos y en México) y establece el número de caracteres decimales en 2, de manera predeterminada.
D o d	Da formato al objeto string como decimal. Muestra el número como un entero.
N o n	Da formato al objeto string con comas y un valor predeterminado de dos lugares decimales.
E o e	Da formato al objeto string utilizando notación científica, con un valor predeterminado de seis lugares decimales.
F o f	Da formato al objeto string con un número fijo de lugares decimales (el valor predeterminado es de dos).
G o g	General. Por lo general da formato al número con lugares decimales o utilizando notación científica, dependiendo del contexto. Si un elemento de formato no contiene un especificador de formato, se asume el formato G de manera implícita.
X o X	Da formato al objeto string como hexadecimal.

**Figura 4.17** | Especificadores de formato para objetos `string`.

Las configuraciones culturales en el equipo del usuario determinan el formato para mostrar cantidades monetarias. Por ejemplo, en Estados Unidos, 50 se muestra como \$50.00; mientras que en Alemania, como 50,00€. La figura 4.17 muestra algunos especificadores de formato, además de C.

La línea 19 declara la variable local `montoDeposito` para almacenar cada monto a depositar que escriba el usuario. A diferencia de la variable de instancia `saldo` en la clase `Cuenta`, la variable local `montoDeposito` en `Main` no se inicializa con 0 de manera predeterminada. No obstante, esta variable no necesita inicializarse aquí, ya que su valor se determinará en base a la entrada del usuario.

La línea 22 pide al usuario que introduzca un monto a depositar para `cuenta1`. La línea 23 obtiene la entrada del usuario mediante una llamada al método `ReadLine` de la clase `Console`, y después pasa el objeto `string` que introdujo el usuario al método `ToDecimal` de la clase `Convert`, el cual devuelve el valor `decimal` en este objeto `string`. Las líneas 24-25 muestran en pantalla el monto de depósito. La línea 26 llama al método `Acredita` del objeto `cuenta1` y suministra `montoDeposito` como argumento para el método. Cuando se hace la llamada al método, el valor del argumento se asigna al parámetro `monto` del método `Acredita` (líneas 16-19 de la figura 4.15) y después este método suma ese valor al `saldo` (línea 18 de la figura 4.15). Las líneas 29-32 (figura 4.16) imprimen en pantalla los saldos de ambos objetos `Cuenta` otra vez, para mostrar que sólo cambió el saldo de `cuenta1`.

La línea 35 pide al usuario que introduzca un monto a depositar en `cuenta2`. La línea 36 obtiene la entrada del usuario mediante una llamada al método `ReadLine` de la clase `Console` y pasa el valor de retorno al método `ToDecimal` de la clase `Convert`. Las líneas 37-38 muestran en pantalla el monto de depósito. La línea 39 llama al método `Acredita` del objeto `cuenta2` y suministra `montoDeposito` como argumento para el método, después el método `Acredita` suma ese valor al saldo. Por último, las líneas 42-43 imprimen en pantalla los saldos de ambos objetos `Cuenta` otra vez, para mostrar que sólo cambió el saldo de `cuenta2`.

#### **Descriptores `set` y `get` con distintos modificadores de acceso**

De manera predeterminada, los descriptores de acceso `get` y `set` de una propiedad tienen el mismo acceso que la propiedad; por ejemplo, para una propiedad `public`, los descriptores de acceso son `public`. Es posible declarar los descriptores de acceso `get` y `set` con distintos modificadores de acceso. En este caso, uno de los descriptores de acceso debe tener de manera implícita el mismo acceso que la propiedad, y el otro debe declararse con un modificador de acceso más restrictivo que el de la propiedad. Por ejemplo, en una propiedad `public`, el descriptor de acceso `get` podría ser `public` y el descriptor de acceso `set` podría ser `private`. En la sección 9.6 demostraríamos esta característica.



### Tip de prevención de errores 4.2

*Los beneficios de la integridad de datos no son automáticos sólo porque las variables de instancia sean private; usted debe proporcionar una comprobación de validez apropiada y reportar los errores.*



### Tip de prevención de errores 4.3

*Los descriptores de acceso set que establecen los valores de datos private deben verificar que los nuevos valores deseados sean apropiados; si no lo son, las variables de instancia deben permanecer sin cambio y se genera un error. En el capítulo 12, Manejo de excepciones, demostraremos cómo generar errores con sutileza.*

#### Diagrama de clases de UML para la clase Cuenta

El diagrama de clases de UML en la figura 4.18 modela la clase *Cuenta* de la figura 4.15. El diagrama modela la propiedad *Saldo* como un atributo de UML de tipo *decimal* (ya que la propiedad correspondiente en C# tiene el tipo *decimal*). El diagrama modela el constructor de la clase *Cuenta* con un parámetro *saldoinicial* de tipo *decimal* en el tercer compartimiento de la clase. El diagrama modela la operación *Acredita* en el tercer compartimiento con un parámetro *monto* de tipo *decimal* (ya que el método correspondiente tiene un parámetro *monto* de tipo *decimal* en C#).

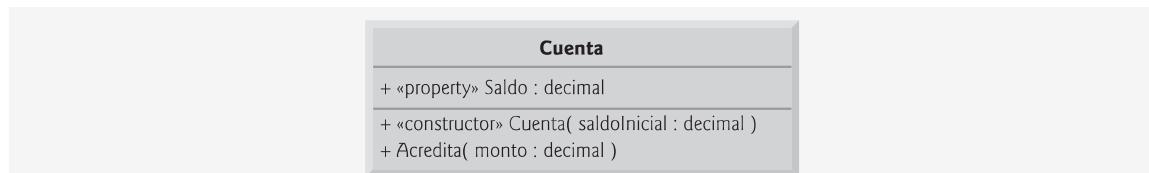
## 4.11 (Opcional) Caso de estudio de ingeniería de software: identificación de las clases en el documento de requerimientos del ATM

Ahora empezaremos a diseñar el sistema ATM que presentamos en el capítulo 3. En esta sección identificaremos las clases necesarias para crear el sistema ATM; para ello es necesario analizar los sustantivos y las frases nominales que aparecen en el documento de requerimientos. Introducimos los diagramas de clases de UML para modelar las relaciones entre estas clases. Este primer paso es importante para definir la estructura de nuestro sistema.

#### Identificación de las clases en un sistema

Para comenzar nuestro proceso de diseño orientado a objetos (OO), identificaremos las clases requeridas para crear el sistema ATM. Más adelante describiremos estas clases mediante el uso de los diagramas de clases de UML y las implementaremos en C#. Primero debemos revisar el documento de requerimientos de la sección 3.10, para encontrar sustantivos y frases nominales clave que nos ayuden a identificar las clases que conformarán el sistema ATM. Tal vez decidamos que algunos de estos sustantivos y frases nominales sean atributos de otras clases en el sistema. Tal vez también concluyamos que algunos de los sustantivos y frases nominales no corresponden a ciertas partes del sistema y, por ende, no deben modelarse. A medida que avancemos por el proceso de diseño podemos ir descubriendo clases adicionales. La figura 4.19 lista los sustantivos y frases nominales en el documento de requerimientos.

Crearemos clases sólo para los sustantivos y frases nominales que tengan importancia en el sistema ATM. No necesitamos modelar “banco” como una clase, ya que el banco no es una parte del sistema ATM; el banco sólo quiere que nosotros construyamos el ATM. “Usuario” y “cliente” también representan entidades fuera del sistema; son importantes debido a que interactúan con nuestro sistema ATM, pero no necesitamos modelarlos como clases en el sistema ATM. Recuerde que modelamos un usuario del ATM (es decir, un cliente bancario) como el actor en el diagrama de caso-uso de la figura 3.31.



**Figura 4.18** | Diagrama de clases de UML que indica que la clase *Cuenta* tiene una propiedad *public* llamada *Saldo* de tipo *decimal*, un constructor y un método.

Sustantivos y frases nominales en el documento de requerimientos		
banco	dinero / fondos	número de cuenta
ATM	pantalla	NIP
usuario	teclado numérico	base de datos del banco
cliente	dispensador de efectivo	solicitud de saldo
transacción	billete de \$20 / efectivo	retiro
cuenta	ranura de depósito	depósito
saldo	sobre de depósito	

**Figura 4.19** | Sustantivos y frases nominales en el documento de requerimientos.

No necesitamos modelar “billete de \$20” ni “sobre de depósito” como clases. Éstos son objetos físicos en el mundo real, pero no forman parte de lo que se automatizará. Podemos representar en forma adecuada la presencia de billetes en el sistema, mediante el uso de un atributo de la clase que modela el dispensador de efectivo (en la sección 5.12 asignaremos atributos a las clases). Por ejemplo, el dispensador de efectivo mantiene un conteo del número de billetes que contiene. El documento de requerimientos no dice nada acerca de lo que debe hacer el sistema con los sobres de depósito después de recibirlos. Podemos suponer que con sólo admitir la recepción de un sobre (una **operación** que realiza la clase que modela la ranura de depósito) es suficiente para representar la presencia de un sobre en el sistema (en la sección 7.15 asignaremos operaciones a las clases).

En nuestro sistema ATM simplificado, lo más apropiado sería representar varios montos de “dinero”, incluyendo el “saldo” de una cuenta, como atributos de otras clases. De igual forma, los sustantivos “número de cuenta” y “NIP” representan piezas importantes de información en el sistema ATM. Son atributos importantes de una cuenta bancaria. Sin embargo, no ofrecen comportamientos. Por ende, podemos modelarlos de la manera más apropiada como atributos de una clase de cuenta.

Aunque con frecuencia el documento de requerimientos describe una “transacción” en un sentido general, no modelaremos la amplia noción de una transacción financiera en este momento. En vez de ello, modelaremos los tres tipos de transacciones (es decir, “solicitud de saldo”, “retiro” y “depósito”) como clases individuales. Estas clases poseen los atributos específicos necesarios para ejecutar las transacciones que representan. Por ejemplo, para un retiro se necesita conocer el monto de dinero que el usuario desea retirar. Sin embargo, una solicitud de saldo no requiere datos adicionales. Lo que es más, las tres clases de transacciones exhiben comportamientos únicos. Para un retiro se requiere entregar efectivo al usuario, mientras que para un depósito se requiere recibir un sobre de depósito del usuario. [Nota: en la sección 11.9, “factorizaremos” las características comunes de todas las transacciones en una clase de “transacción” general, mediante el uso de los conceptos orientados a objetos de las clases abstractas y la herencia.]

Determinaremos las clases para nuestro sistema con base en los sustantivos y frases nominales restantes de la figura 4.19. Cada una de ellas se refiere a uno o varios de los siguientes elementos:

- ATM.
- pantalla.
- teclado numérico.
- dispensador de efectivo.
- ranura de depósito.
- cuenta.
- base de datos del banco.
- solicitud de saldo.
- retiro.
- depósito.

Es probable que los elementos de esta lista sean clases que necesitaremos implementar en nuestro sistema, aunque es demasiado pronto en nuestro proceso de diseño como para decir que la lista está completa.

Ahora podemos modelar las clases en nuestro sistema, con base en la lista que hemos creado. En el proceso de diseño escribimos los nombres de las clases con la primera letra en mayúscula (una convención de UML), como lo haremos cuando escribamos el código de C# para implementar nuestro diseño. Si el nombre de una clase contiene más de una palabra, juntaremos todas las palabras y escribiremos la primera letra de cada una de ellas en mayúscula (por ejemplo, *NombreConVariasPalabras*). Con estas convenciones, crearemos las clases ATM, Pantalla, Teclado, DispensadorEfectivo, RanuraDeposito, Cuenta, BaseDatosBanco, SolicitudSaldo, Retiro y Deposito. Construiremos nuestro sistema mediante el uso de todas estas clases como bloques de construcción. Sin embargo, antes de empezar a construir el sistema, debemos comprender mejor la forma en que las clases se relacionan entre sí.

### **Modelado de las clases**

UML nos permite modelar, a través de los **diagramas de clases**, las clases en el sistema ATM y sus interrelaciones. La figura 4.20 representa a la clase ATM. En UML, cada clase se modela como un rectángulo con tres compartimentos. El compartimiento superior contiene el nombre de la clase, centrado en forma horizontal y en negrita. El compartimiento de en medio contiene los atributos de la clase (en las secciones 5.12 y 6.9 hablaremos sobre los atributos). El compartimiento inferior contiene las operaciones de la clase (que veremos en la sección 7.15). En la figura 4.20 los compartimientos de en medio e inferior están vacíos, ya que no hemos determinado los atributos y operaciones de esta clase todavía.

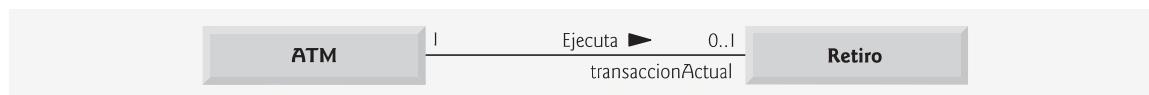
Los diagramas de clases también muestran las relaciones entre las clases del sistema. La figura 4.21 muestra cómo nuestras clases ATM y Retiro se relacionan una con la otra. Por el momento modelaremos sólo este subconjunto de las clases del ATM, por cuestión de simpleza. Más adelante en esta sección, presentaremos un diagrama de clases más completo. Observe que los rectángulos que representan a las clases en este sistema no están subdivididos en compartimentos. UML permite suprimir los atributos y las operaciones de una clase de esta forma, cuando sea apropiado, para crear diagramas más legibles. Un diagrama de este tipo se denomina **diagrama con elementos omitidos (elided diagram)**: tanto su información como el contenido de los compartimientos segundo y tercero, no se modela. En las secciones 5.12 y 7.15 colocaremos información en estos compartimientos.

En la figura 4.21, la línea sólida que conecta a las dos clases representa una **asociación**: una relación entre clases. Los números cerca de cada extremo de la línea son valores de **multiplicidad**; éstos indican cuántos objetos de cada clase participan en la asociación. En este caso, al seguir la línea de un extremo al otro se revela que, en un momento dado, un objeto ATM participa en una asociación con cero o con un objeto Retiro; cero si el usuario actual no está realizando una transacción o si ha solicitado un tipo distinto de transacción, y uno si el usuario ha solicitado un retiro. UML puede modelar muchos tipos de multiplicidad. La figura 4.22 explica los tipos de multiplicidad.

Una asociación puede tener nombre. Por ejemplo, la palabra Ejecuta por encima de la línea que conecta a las clases ATM y Retiro en la figura 4.21 indica el nombre de esa asociación. Esta parte del diagrama se lee así: “un objeto de la clase ATM ejecuta cero o un objeto de la clase Retiro”. Los nombres de las asociaciones son direc-



**Figura 4.20** | Representación de una clase en UML mediante un diagrama de clases.



**Figura 4.21** | Diagrama de clases que muestra una asociación entre clases.

Símbolo	Descripción
0	Ninguno
1	Uno
$m$	Un valor entero
0..1	Cero o uno
$m, n$	$m$ o $n$
$m..n$	Cuando menos $m$ , pero no más que $n$
*	Cualquier entero no negativo (cero o más)
0..*	Cero o más (idéntico a *)
1..*	Uno o más

Figura 4.22 | Tipos de multiplicidad.

cionales, como lo indica la punta de flecha rellena; por lo tanto, sería inapropiado, por ejemplo, leer la anterior asociación de derecha a izquierda como “cero o un objeto de la clase `Retiro` ejecuta un objeto de la clase `ATM`”.

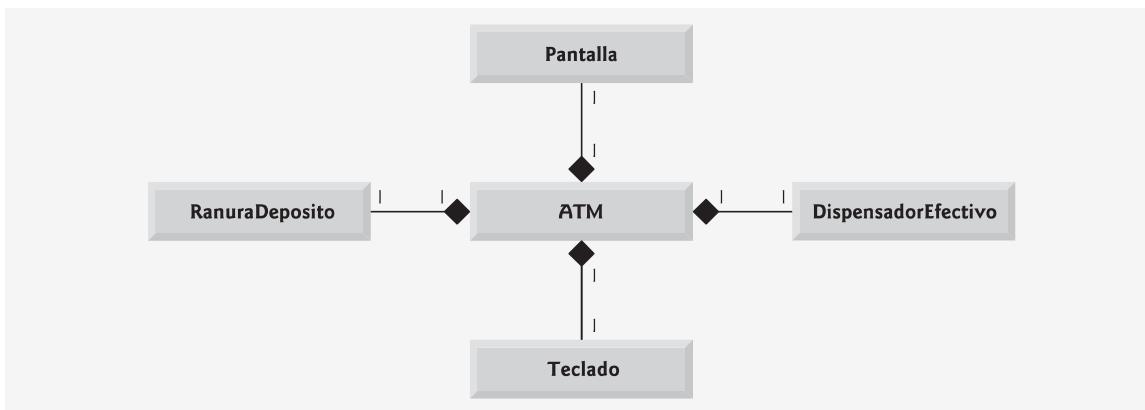
La palabra `transaccionActual` en el extremo de `Retiro` de la línea de asociación en la figura 4.21 es un **nombre de rol**, el cual identifica el rol que desempeña el objeto `Retiro` en su relación con el `ATM`. Un nombre de rol agrega significado a una asociación entre clases, ya que identifica el rol que desempeña una clase dentro del contexto de una asociación. Una clase puede desempeñar varios roles en el mismo sistema. Por ejemplo, en un sistema de personal de una universidad, una persona puede desempeñar el rol de “profesor” con respecto a los estudiantes. La misma persona puede desempeñar el rol de “colega” cuando participa en una relación con otro profesor, y de “entrenador” cuando entrena a los atletas estudiantes. En la figura 4.21, el nombre de rol `transaccionActual` indica que el objeto `Retiro` que participa en la asociación `Ejecuta` con un objeto de la clase `ATM` representa a la transacción que está procesando el `ATM` en ese momento. En otros contextos, un objeto `Retiro` puede desempeñar otros roles (por ejemplo, la transacción anterior). Observe que no especificamos un nombre de rol para el extremo del `ATM` de la asociación `Ejecuta`. A menudo, los nombres de los roles se omiten en los diagramas de clases cuando el significado de una asociación está claro sin ellos.

Además de indicar relaciones simples, las asociaciones pueden especificar relaciones más complejas, como cuando los objetos de una clase están compuestos de objetos de otras clases. Considere un cajero automático real. ¿Qué “piezas” reúne un fabricante para construir un `ATM` funcional? Nuestro documento de requerimientos nos indica que el `ATM` está compuesto de una pantalla, un teclado, un dispensador de efectivo y una ranura de depósito.

En la figura 4.23, los **diamantes sólidos** que se adjuntan a las líneas de asociación de la clase `ATM` indican que esta clase tiene una relación de **composición** con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`. La composición implica una relación en todo/en parte. La clase que tiene el símbolo de composición (el diamante sólido) en su extremo de la línea de asociación es el todo (en este caso, `ATM`) y las clases en el otro extremo de las líneas de asociación son las partes; en este caso, las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`. Las composiciones en la figura 4.23 indican que un objeto de la clase `ATM` está formado de un objeto de la clase `Pantalla`, un objeto de la clase `DispensadorEfectivo`, un objeto de la clase `Teclado` y un objeto de la clase `RanuraDeposito`; el `ATM` “tiene una” pantalla, un teclado, un dispensador de efectivo y una ranura de depósito. La **relación “tiene un”** define la composición (en la sección del Caso de estudio de ingeniería de software del capítulo 11 veremos que la relación “es un” define la herencia).

De acuerdo con la especificación del UML, las relaciones de composición tienen las siguientes propiedades:

1. Sólo una clase en la relación puede representar el todo (es decir, el diamante puede colocarse sólo en un extremo de la línea de asociación). Por ejemplo, la pantalla es parte del `ATM` o el `ATM` es parte de la pantalla, pero la pantalla y el `ATM` no pueden representar ambos el todo dentro de la relación.
2. Las partes en la relación de composición existen sólo mientras exista el todo, y el todo es responsable de la creación y destrucción de sus partes. Por ejemplo, el acto de construir un `ATM` incluye la manufac-



**Figura 4.23** | Diagrama de clases que muestra las relaciones de composición.

ra de sus partes. Lo que es más, si el ATM se destruye, también se destruyen su pantalla, teclado, dispensador de efectivo y ranura de depósito.

3. Una parte puede pertenecer sólo a un todo a la vez, aunque esa parte puede quitarse y unirse a otro todo, el cual entonces asumirá la responsabilidad de esa parte.

Los diamantes sólidos en nuestros diagramas de clases indican las relaciones de composición que cumplen con estas tres propiedades. Si una relación “es un” no satisface uno o más de estos criterios, UML especifica que se deben adjuntar diamantes sin relleno a los extremos de las líneas de asociación para indicar una **agregación**: una forma más débil de la composición. Por ejemplo, una computadora personal y un monitor de computadora participan en una relación de agregación: la computadora “tiene un” monitor, pero las dos partes pueden existir en forma independiente, y el mismo monitor puede conectarse a varias computadoras a la vez, con lo cual se violan las propiedades segunda y tercera de la composición.

La figura 4.24 muestra un diagrama de clases para el sistema ATM. Este diagrama modela la mayoría de las clases que identificamos antes en esta sección, así como las asociaciones entre ellas que podemos inferir del documento de requerimientos. [Nota: las clases **SolicitudSaldo** y **Depósito** participan en asociaciones similares a las de la clase **Retiro**, por lo que preferimos omitirlas en este diagrama por cuestión de simpleza. En el capítulo 11 expandiremos nuestro diagrama de clases para incluir todas las clases en el sistema ATM.]

La figura 4.24 presenta un modelo gráfico de la estructura del sistema ATM. Este diagrama de clases incluye las clases **BaseDatosBanco** y **Cuenta**, junto con varias asociaciones que no presentamos en las figuras 4.21 o 4.23. El diagrama de clases muestra que la clase **ATM** tiene una **relación de uno a uno** con la clase **BaseDatosBanco**: un objeto ATM autentica a los usuarios contra un objeto **BaseDatosBanco**. En la figura 4.24 también modelamos el hecho de que la base de datos del banco contiene información sobre muchas cuentas; un objeto de la clase **BaseDatosBanco** participa en una relación de composición con cero o más objetos de la clase **Cuenta**. Recuerde que en la figura 4.22 se muestra que el valor de multiplicidad 0..\* en el extremo de la clase **Cuenta** de la asociación entre las clases **BaseDatosBanco** y **Cuenta** indica que cero o más objetos de la clase **Cuenta** participan en la asociación. La clase **BaseDatosBanco** tiene una **relación de uno a varios** con la clase **Cuenta**; **BaseDatosBanco** puede contener muchos objetos **Cuenta**. De manera similar, la clase **Cuenta** tiene una **relación de varios a uno** con la clase **BaseDatosBanco**; puede haber muchos objetos **Cuenta** en **BaseDatosBanco**. Si recuerda la figura 4.22, el valor de multiplicidad \* es idéntico a 0..\*.

La figura 4.24 también indica que si el usuario va a realizar un retiro, “un objeto de la clase **Retiro** accede a/modifica un saldo de cuenta a través de un objeto de la clase **BaseDatosBanco**”. Podríamos haber creado una asociación en forma directa entre la clase **Retiro** y la clase **Cuenta**. No obstante, el documento de requerimientos indica que el “ATM debe interactuar con la base de datos de información de las cuentas del banco” para realizar transacciones. Una cuenta de banco contiene información delicada, por lo que los ingenieros de sistemas deben considerar siempre la seguridad de los datos personales al diseñar un sistema. Por ello, sólo **BaseDatosBanco** puede acceder a una cuenta y manipularla en forma directa. Todas las demás partes del sistema deben

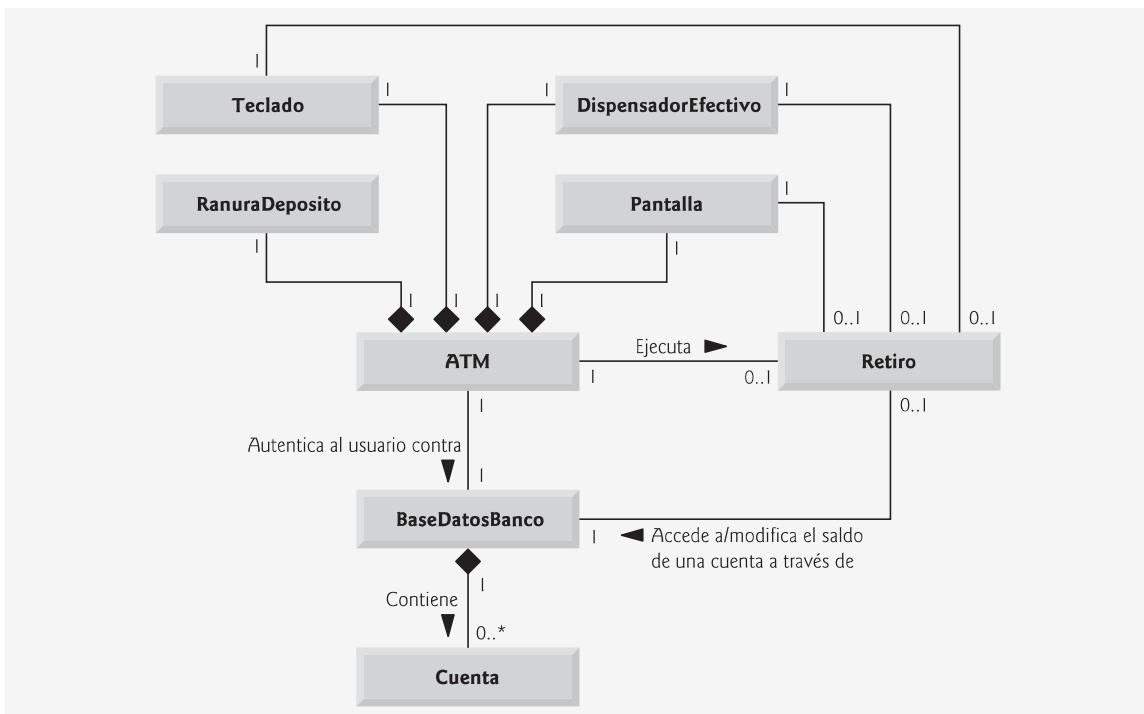


Figura 4.24 | Diagrama de clases para el modelo del sistema ATM.

interactuar con la base de datos para recuperar o actualizar la información de las cuentas (por ejemplo, el saldo de una cuenta).

El diagrama de clases de la figura 4.24 también modela las asociaciones entre la clase **Retiro** y las clases **Pantalla**, **DispensadorEfectivo** y **Teclado**. Una transacción de retiro implica pedir al usuario que seleccione el monto a retirar; también implica recibir entrada numérica. Estas acciones requieren el uso de la pantalla y del teclado, en forma respectiva. Para entregar efectivo al usuario se requiere acceso al dispensador de efectivo.

Aunque no se muestran en la figura 4.24, las clases **SolicitudSaldo** y **Deposito** participan en varias asociaciones con las otras clases del sistema ATM. Al igual que la clase **Retiro**, cada una de estas clases se asocia con las clases **ATM** y **BaseDatosBanco**. Un objeto de la clase **SolicitudSaldo** también se asocia con un objeto de la clase **Pantalla** para mostrar al usuario el saldo de una cuenta. La clase **Deposito** se asocia con las clases **Pantalla**, **Teclado** y **RanuraDeposito**. Al igual que los retiros, las transacciones de depósito requieren el uso de la pantalla y el teclado para mostrar mensajes y recibir entradas, en forma respectiva. Para recibir un sobre de depósito, un objeto de la clase **Deposito** se asocia con un objeto de la clase **RanuraDeposito**.

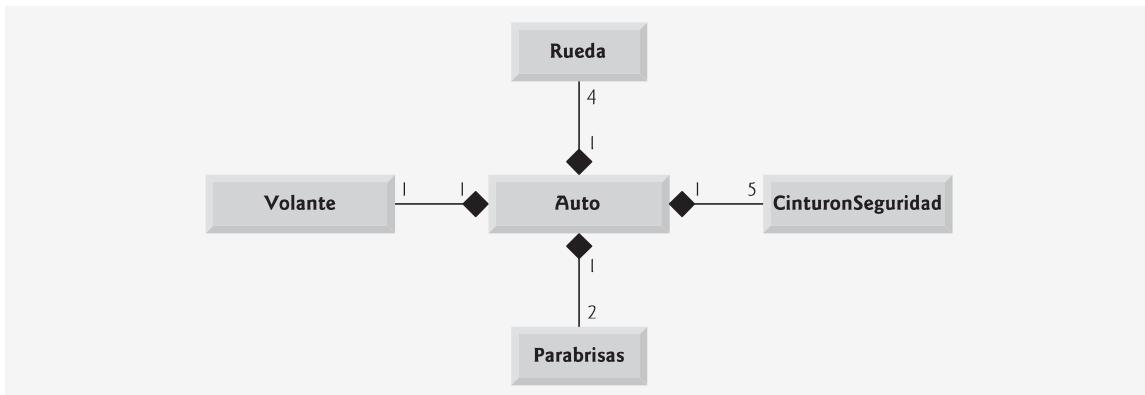
Hemos identificado las clases en nuestro sistema ATM, aunque tal vez descubramos otras a medida que avancemos con el diseño y la implementación. En la sección 5.12 determinaremos los atributos para cada una de estas clases, y en la sección 6.9 utilizaremos estos atributos para examinar la forma en que cambia el sistema con el tiempo. En la sección 7.15 determinaremos las operaciones de las clases en nuestro sistema.

### Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

4.1 Suponga que tenemos una clase llamada **Auto**, la cual representa a un auto. Piense en algunas de las distintas piezas que podría reunir un fabricante para producir un auto completo. Cree un diagrama de clases (similar a la figura 4.23) que modele algunas de las relaciones de composición de la clase **Auto**.

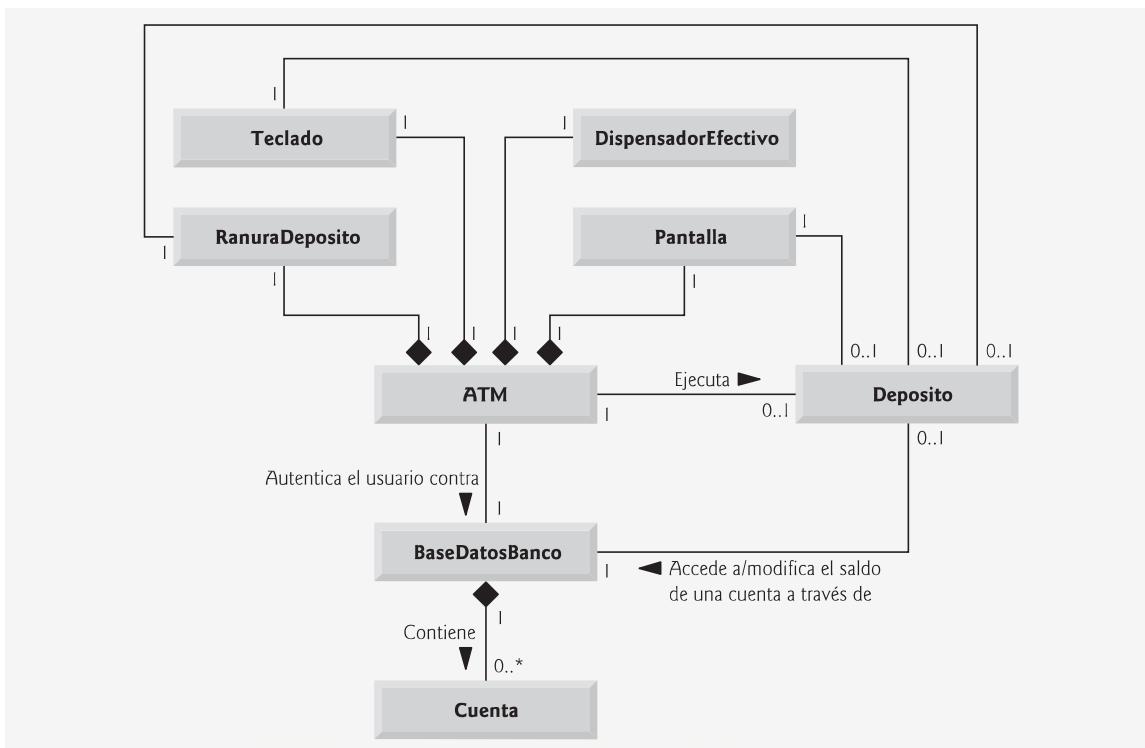
4.2 Suponga que tenemos una clase llamada **Archivo**, la cual representa un documento electrónico en una computadora independiente, sin conexión de red, representada por la clase **Computadora**. ¿Qué tipo de asociación existe entre la clase **Computadora** y la clase **Archivo**?

- La clase **Computadora** tiene una relación de uno a uno con la clase **Archivo**.



**Figura 4.25** | Diagrama de clases que muestra algunas relaciones de composición de una clase **Auto**.

- b) La clase **Computadora** tiene una relación de varios a uno con la clase **Archivo**.  
 c) La clase **Computadora** tiene una relación de uno a varios con la clase **Archivo**.  
 d) La clase **Computadora** tiene una relación de varios a varios con la clase **Archivo**.
- 4.3** Indique si la siguiente aseveración es *verdadera* o *falsa*. Si es *falsa*, explique por qué: un diagrama de clases de UML en el que no se modelan los compartimientos segundo y tercero se denomina diagrama con elementos omitidos (elided diagram).
- 4.4** Modifique el diagrama de clases de la figura 4.24 para incluir la clase **Deposito**, en vez de la clase **Retiro**.



**Figura 4.26** | Diagrama de clases para el modelo del sistema ATM, incluyendo la clase **Deposito**.

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

**4.1** La figura 4.25 presenta un diagrama de clases que muestra algunas de las relaciones de composición de una clase Auto.

**4.2** c. En una computadora con conexión de red, esta relación podría ser de varios a varios.

**4.3** Verdadera.

**4.4** La figura 4.26 presenta un diagrama de clases para el ATM, en el cual se incluye la clase Deposito en vez de la clase Retiro (como en la figura 4.24). Observe que la clase Deposito no se asocia con la clase DispensadorEfectivo, sino que se asocia con la clase RanuraDeposito.

## 4.12 Conclusión

En este capítulo aprendió los conceptos básicos orientados a objetos de las clases, los objetos, métodos, las variables de instancia y las propiedades; utilizaremos estos conceptos en aplicaciones de C# más robustas que crearemos. Aprendió a declarar variables de instancia de una clase para mantener los datos de cada objeto de la clase, a declarar métodos que operan sobre esos datos y cómo declarar propiedades para obtener y establecer esos datos. Demostramos cómo llamar a un método para decirle que realice su tarea y cómo pasar información a los métodos en forma de argumentos. Vimos la diferencia entre una variable local de un método y una variable de instancia de una clase, y que sólo las variables de instancia se inicializan en forma automática. También aprendió a utilizar el constructor de una clase para especificar los valores iniciales para las variables de instancia de un objeto. Vimos algunas de las diferencias entre los tipos por valor y los tipos por referencia. Aprendió acerca de los tipos por valor `float`, `double` y `decimal` para almacenar números reales.

A lo largo de este capítulo le mostramos cómo puede utilizarse el UML para crear diagramas de clases que modelen los constructores, métodos, propiedades y atributos de las clases. Aprendió el valor de declarar las variables de instancia como `private`, y a utilizar las propiedades `public` para manipularlas. Por ejemplo, demostramos cómo los descriptores de acceso `set` en las propiedades pueden utilizarse para validar los datos de un objeto y asegurar que ese objeto se mantenga en un estado consistente. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de una aplicación. Utilizará estas instrucciones en sus métodos para especificar cómo deben realizar sus tareas.