



Polimorfismo, interfaces y sobrecarga de operadores

OBJETIVOS

En este capítulo aprenderá lo siguiente:

- El concepto de polimorfismo y cómo le permite “programar en general”.
- Cómo utilizar métodos redefinidos para efectuar el polimorfismo.
- Diferenciar entre las clases concretas y abstractas.
- Declarar métodos abstractos para crear clases abstractas.
- Cómo el poliformismo hace que los sistemas sean extensibles y administrables.
- Determinar el tipo de un objeto en tiempo de ejecución.
- Crear métodos y clases `sealed`.
- Declarar e implementar interfaces.
- Sobrecargar operadores para permitirles manipular objetos.

Un anillo para gobernarlos a todos, un anillo para encontrarlos, un anillo para traerlos a todos y en la oscuridad enlazarlos.

—John Ronald Reuel Tolkien

Las proposiciones generales no deciden casos concretos.

—Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en un vacío. Incluso, sus ideas más abstractas son, en cierta medida, condicionadas por lo que se conoce o no en el tiempo en que vive.

—Alfred North Whitehead

¿Por qué, alma mía, desfalleces y te agitas por mí?

—Salmos 42:5

Plan general

- 11.1 Introducción
- 11.2 Ejemplos de polimorfismo
- 11.3 Demostración del comportamiento polimórfico
- 11.4 Clases y métodos abstractos
- 11.5 Caso de estudio: sistema de nómina utilizando polimorfismo
 - 11.5.1 Creación de la clase base abstracta `Empleado`
 - 11.5.2 Creación de la clase derivada concreta `EmpleadoAsalariado`
 - 11.5.3 Creación de la clase derivada concreta `EmpleadoPorHoras`
 - 11.5.4 Creación de la clase derivada concreta `EmpleadoPorComision`
 - 11.5.5 Creación de la clase derivada concreta indirecta `EmpleadoBaseMasComision`
 - 11.5.6 El procesamiento polimórfico, el operador `is` y la conversión descendente
 - 11.5.7 Resumen de las asignaciones permitidas entre variables de la clase base y de la clase derivada
- 11.6 Métodos y clases `sealed`
- 11.7 Caso de estudio: creación y uso de interfaces
 - 11.7.1 Desarrollo de una jerarquía `IPorPagar`
 - 11.7.2 Declaración de la interfaz `IPorPagar`
 - 11.7.3 Creación de la clase `Factura`
 - 11.7.4 Modificación de la clase `Empleado` para implementar la interfaz `IPorPagar`
 - 11.7.5 Modificación de la clase `EmpleadoAsalariado` para usarla en la jerarquía `IPorPagar`
 - 11.7.6 Uso de la interfaz `IPorPagar` para procesar objetos `Factura` y `Empleado` mediante el polimorfismo
 - 11.7.7 Interfaces comunes de la Biblioteca de clases del .NET Framework
- 11.8 Sobre carga de operadores
- 11.9 (Opcional) Caso de estudio de ingeniería de software: incorporación de la herencia y el polimorfismo en el sistema ATM
- 11.10 Conclusión

11.1 Introducción

Ahora continuaremos nuestro estudio de la programación orientada a objetos, explicando y demostrando el *polimorfismo* con las jerarquías de herencia. El polimorfismo nos permite “programar en forma general”, en vez de “programar en forma específica”. En especial, nos permite escribir aplicaciones que procesen objetos de clases que formen parte de la misma jerarquía de clases, como si todos fueran objetos de la clase.

Considere el siguiente ejemplo de polimorfismo. Suponga que crearemos una aplicación que simula el movimiento de varios tipos de animales para un estudio biológico. Las clases `Pez`, `Rana` y `Ave` representan los tres tipos de animales bajo investigación. Imagine que cada una de estas clases extiende a la clase base `Animal`, que contiene un método llamado `Mover` y mantiene la posición actual de un animal, en forma de coordenadas *x-y*. Cada clase derivada implementa el método `Mover`. Nuestra aplicación mantiene un arreglo de referencias a objetos de las diversas clases derivadas de `Animal`. Para simular los movimientos de los animales, la aplicación envía a cada objeto el mismo mensaje una vez por segundo; a saber, `Mover`. No obstante, cada tipo específico de `Animal` responde a un mensaje `Mover` de manera única; un `Pez` podría nadar tres pies, una `Rana` podría saltar cinco pies y un `Ave` podría volar 10 pies. La aplicación envía el mismo mensaje (es decir, `Mover`) a cada objeto animal en forma genérica, pero cada objeto sabe cómo modificar sus coordenadas *x-y* en forma apropiada para su tipo específico de movimiento. Confiar en que cada objeto sepa cómo “hacer lo correcto” (es decir, lo que sea apropiado para ese tipo de objeto) en respuesta a la llamada al mismo método es el concepto clave del polimorfismo. El mismo mensaje (en este caso, `Mover`) que se envía a una variedad de objetos tiene “muchas formas” de resultados; de aquí que se utilice el término polimorfismo.

Con el polimorfismo podemos diseñar e implementar sistemas que puedan extenderse con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales de la aplicación, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que la aplicación procesa en forma genérica. Las únicas partes de una aplicación que deben alterarse para alojar nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador va a agregar a la jerarquía. Por ejemplo, si extendemos la clase `Animal` para crear la clase `Tortuga` (que podría responder a un mensaje `Mover` caminando una pulgada), necesitamos escribir sólo la clase `Tortuga` y la parte de la simulación que crea una instancia de un objeto `Tortuga`. Las porciones de la simulación que procesan a cada `Animal` en forma genérica pueden permanecer iguales.

Este capítulo se divide en varias partes. Primero hablaremos sobre los ejemplos comunes del polimorfismo. Después proporcionaremos un ejemplo de código activo, en el que se demuestra el comportamiento polimórfico. Como pronto verá, utilizará referencias a la clase base para manipular tanto los objetos de la clase base como los de las clases derivadas mediante el polimorfismo.

Después presentaremos un caso de estudio en el que volveremos a utilizar la jerarquía de empleados de la sección 10.4.5. Desarrollaremos una aplicación simple de nómina que, mediante el polimorfismo, calcula el salario semanal de varios tipos de empleados, usando el método `Ingresos` de cada empleado. Aunque los ingresos de cada tipo de empleado se calculan de una manera específica, el polimorfismo nos permite procesar a los empleados “en general”. En el caso de estudio ampliaremos la jerarquía para incluir dos nuevas clases: `EmpleadoAsalariado` (para las personas que reciben un salario semanal fijo) y `EmpleadoPorHoras` (para las personas que reciben un salario por horas y “tiempo y medio” por el tiempo extra). Declararemos un conjunto común de funcionalidad para todas las clases en la jerarquía actualizada en una clase “abstracta” llamada `Empleado`, a partir de la cual las clases `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` heredan en forma directa, y la clase `EmpleadoBaseMasComision4` hereda en forma indirecta. Como veremos a continuación, cuando invocamos el método `Ingresos` de cada empleado desde una referencia a la clase base `Empleado`, se realiza el cálculo correcto de los ingresos debido a las capacidades polimórficas de C#.

Algunas veces, cuando se lleva a cabo el procesamiento del polimorfismo, es necesario programar “en forma específica”. Nuestro caso de estudio con `Empleado` demuestra que una aplicación puede determinar el tipo de un objeto en tiempo de ejecución, y actuar sobre ese objeto de manera acorde. En el caso de estudio utilizamos estas capacidades para determinar si cierto objeto empleado *es un* `EmpleadoBaseMasComision`. Si es así, incrementamos el salario base de ese empleado en 10 por ciento.

El capítulo continúa con una introducción a las interfaces en C#. Una interfaz describe a un conjunto de métodos y propiedades que pueden llamarse en un objeto, pero no proporciona implementaciones concretas para ellos. Los programadores pueden declarar clases que *implementen* a (es decir, que proporcionen implementaciones concretas para los métodos y propiedades de) una o más interfaces. Cada miembro de interfaz debe declararse en todas las clases que implementen a la interfaz. Una vez que una clase implementa a una interfaz, todos los objetos de esa clase tienen una relación *es un* con el tipo de la interfaz, y se garantiza que todos los objetos de la clase proporcionarán la funcionalidad descrita por la interfaz. Esto se aplica también para todas las clases derivadas de esa clase.

En especial, las interfaces son útiles para asignar la funcionalidad común a clases que posiblemente no estén relacionadas. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de las clases que implementan la misma interfaz pueden responder a las mismas llamadas a los métodos. Para demostrar la creación y el uso de interfaces, modificaremos nuestra aplicación de nómina para crear una aplicación general de cuentas por pagar, que puede calcular los pagos vencidos por los ingresos de los empleados de la compañía y los montos de las facturas a pagar por los bienes comprados. Como verá, las interfaces permiten capacidades polimórficas similares a las que permite la herencia.

Este capítulo termina con una introducción a la sobrecarga de operadores. En capítulos anteriores, declaramos nuestras propias clases y utilizamos métodos para realizar tareas con objetos de esas clases. La sobrecarga de operadores nos permite definir el comportamiento de los operadores integrados, como `+`, `-` y `<`, cuando los utilizamos con objetos de nuestras propias clases. Esto proporciona una notación mucho más conveniente que las llamadas a métodos para realizar tareas con los objetos.

11.2 Ejemplos de polimorfismo

Ahora consideraremos diversos ejemplos adicionales de polimorfismo. Si la clase `Rectangulo` se deriva de la clase `Cuadrilatero` (una figura con cuatro lados), entonces un `Rectangulo` es una versión más específica de

Cuadrilatero. Cualquier operación (por ejemplo, calcular el perímetro o el área) que pueda realizarse en un objeto Cuadrilatero también puede realizarse en un objeto Rectángulo. Estas operaciones también pueden realizarse en otros objetos Cuadrilatero, como Cuadrado, Paralelogramo y Trapezoide. El polimorfismo ocurre cuando una aplicación invoca a un método a través de una variable de la clase base; en tiempo de ejecución, se hace una llamada a la versión correcta del método de la clase derivada, con base en el tipo del objeto referenciado. En la sección 11.3 veremos un ejemplo de código simple, en el cual se ilustra este proceso.

Como otro ejemplo, suponga que vamos a diseñar un videojuego que manipule objetos de muchos tipos distintos, incluyendo objetos de las clases Marciano, Venusino, Plutoniano, NaveEspacial y RayoLaser. Imagine que cada clase hereda de la clase base común ObjetoEspacial, el cual contiene el método Dibujar. Cada clase derivada implementa a este método. Una aplicación de administración de la pantalla mantiene una colección (por ejemplo, un arreglo ObjetoEspacial) de referencias a objetos de las diversas clases. Para refrescar la pantalla, el administrador de pantalla envía en forma periódica el mismo mensaje a cada objeto; a saber, Dibujar. No obstante, cada objeto responde de una manera única. Por ejemplo, un objeto Marciano podría dibujarse a sí mismo en color rojo, con el número apropiado de antenas. Un objeto NaveEspacial podría dibujarse a sí mismo como un platillo volador de color plata brillante. Un objeto RayoLaser podría dibujarse a sí mismo como un rayo color rojo brillante a lo largo de la pantalla. De nuevo, el mismo mensaje (en este caso, Dibujar) que se envía a una variedad de objetos tiene “muchas formas” de resultados.

Un administrador de pantalla polimórfico podría utilizar el polimorfismo para facilitar el proceso de agregar nuevas clases a un sistema, con el menor número de modificaciones al código del sistema. Suponga que deseamos agregar objetos Mercuriano a nuestro videojuego. Para ello, debemos crear una clase Mercuriano que extienda a ObjetoEspacial y proporcione su propia implementación del método Dibujar. Cuando aparezcan objetos de la clase Mercuriano en la colección ObjetoEspacial, el código del administrador de pantalla invocará al método Dibujar, de la misma forma que para cualquier otro objeto en la colección, sin importar su tipo. Por lo tanto, los nuevos objetos Mercuriano simplemente se integran al videojuego sin necesidad de que el programador modifique el código del administrador de pantalla. Así, sin modificar el sistema (más que para crear nuevas clases y modificar el código que genera nuevos objetos), los programadores pueden utilizar el polimorfismo para incluir tipos adicionales que no se hayan considerado a la hora de crear el sistema.



Observación de ingeniería de software 11.1

El polimorfismo promueve la extensibilidad: el software que invoca el comportamiento polimórfico es independiente de los tipos de los objetos a los cuales se envían los mensajes. Se pueden incorporar nuevos tipos de objetos que pueden responder a las llamadas de los métodos existentes, sin necesidad de modificar el sistema base. Sólo el código cliente que crea instancias de los nuevos objetos debe modificarse para dar cabida a los nuevos tipos.

11.3 Demostración del comportamiento polimórfico

En la sección 10.4 creamos una jerarquía de clases de empleados por comisión, en la cual la clase EmpleadoBaseMasComision heredó de la clase EmpleadoPorComision. Los ejemplos en esa sección manipulan objetos EmpleadoPorComision y EmpleadoBaseMasComision mediante el uso de referencias a ellos para invocar a sus métodos. Dirigimos las referencias a la clase base a los objetos de la clase base, y las referencias a la clase derivada a los objetos de la clase derivada. Estas asignaciones son naturales y directas; las referencias a la clase base están diseñadas para referirse a objetos de la clase base, y las referencias a la clase derivada están diseñadas para referirse a objetos de la clase derivada. No obstante, es posible realizar otras asignaciones.

En el siguiente ejemplo, dirigiremos una referencia a la clase base a un objeto de la clase derivada. Después mostraremos cómo al invocar un método en un objeto de la clase derivada a través de una referencia a la clase base se invoca a la funcionalidad de la clase derivada; el tipo del *objeto actual al que se hace referencia*, no el tipo de *referencia*, es el que determina cuál método se va a llamar. Este ejemplo demuestra el concepto clave de que un objeto de una clase derivada puede tratarse como un objeto de su clase base. Esto permite varias manipulaciones interesantes. Una aplicación puede crear un arreglo de referencias a la clase base, que se refieran a objetos de muchos tipos de clases derivadas. Esto se permite, ya que cada objeto de clase derivada *es un* objeto de su clase base. Por ejemplo, podemos asignar la referencia de un objeto EmpleadoBaseMasComision a una variable EmpleadoPorComision de la clase base, ya que un EmpleadoBasePorComision *es un* EmpleadoPorComision; por lo tanto, podemos tratar a un EmpleadoBasePorComision como un EmpleadoPorComision.

Un objeto de la clase base no es un objeto de ninguna de sus clases derivadas. Por ejemplo, no podemos asignar la referencia de un objeto `EmpleadoPorComision` a una variable de la clase derivada `EmpleadoBaseMasComision`, ya que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`; por ejemplo, un `EmpleadoPorComision` no tiene una variable de instancia `salarioBase` y no tiene una propiedad `SalarioBase`. La relación *es un* se aplica de una clase derivada a sus clases base directa e indirecta, pero no viceversa.

Resulta ser que el compilador permite la asignación de una referencia a la clase base a una variable de la clase derivada, si convertimos explícitamente la referencia a la clase base al tipo de la clase derivada; una técnica que veremos con más detalle en la sección 11.5.6. ¿Para qué nos serviría, en un momento dado, realizar una asignación así? Una referencia a la clase base puede usarse para invocar sólo a los métodos declarados en la clase base; si tratamos de invocar métodos que sólo pertenezcan a la clase derivada a través de una referencia a la clase base se producen errores de compilación. Si una aplicación necesita realizar una operación específica para la clase derivada en un objeto de la clase derivada al que se haga una referencia mediante una variable de la clase base, la aplicación primero debe convertir la referencia a la clase base en una referencia a la clase derivada, mediante una técnica conocida como *conversión descendente*. Esto permite a la aplicación invocar métodos de la clase derivada que no se encuentren en la clase base. En la sección 11.5.6 presentaremos un ejemplo concreto de conversión descendente.

El ejemplo de la figura 11.1 demuestra tres formas de usar variables de la clase base y la clase derivada para almacenar referencias a objetos de la clase base y de la clase derivada. Las primeras dos formas son simples: al igual que en la sección 10.4, asignamos una referencia a la clase base a una variable de la clase base, y asignamos una referencia a la clase derivada a una variable de la clase derivada. Después demostramos la relación entre las clases derivadas y las clases base (es decir, la relación *es un*) mediante la asignación de una referencia a la clase derivada a una variable de la clase base. [Nota: esta aplicación utiliza las clases `EmpleadoPorComision3` y `EmpleadoBaseMasComision4` de las figuras 10.13 y 10.14, respectivamente.]

```

1 // Fig. 11.1: PruebaPolimorfismo.cs
2 // Asignación de referencias a la clase base y la clase derivada a
3 // variables de la clase base y de la clase derivada.
4 using System;
5
6 public class PruebaPolimorfismo
7 {
8     public static void Main( string[] args )
9     {
10         // asigna una referencia a la clase base a una variable de la clase base
11         EmpleadoPorComision3 empleadoPorComision = new EmpleadoPorComision3(
12             "Sue", "Jones", "222-22-2222", 10000.00M, .06M );
13
14         // asigna una referencia a la clase derivada a una variable de la clase derivada
15         EmpleadoBaseMasComision4 empleadoBaseMasComision4 =
16             new EmpleadoBaseMasComision4("Bob", "Lewis",
17                 "333-33-3333", 5000.00M, .04M, 300.00M );
18
19         // invoca a ToString y a Ingresos en el objeto de la clase base,
20         // usando la variable de la clase base
21         Console.WriteLine( "{0} {1}:\n{n{2}}\n{n{3}}: {4:C}\n",
22             "Llama a ToString de EmpleadoPorComision3 con referencia de clase base",
23             "a objeto de clase base", empleadoPorComision.ToString(),
24             "ingresos", empleadoPorComision.Ingresos() );
25
26         // invoca a ToString e Ingresos en objeto de clase derivada,
27         // usando variable de clase derivada
28         Console.WriteLine( "{0} {1}:\n{n{2}}\n{n{3}}: {4:C}\n",
29             "Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase",

```

Figura 11.1 | Asignación de referencias a la clase base y a la clase derivada a variables de la clase base y de la clase derivada. (Parte 1 de 2).

```

30      "derivada a objeto de clase derivada",
31      empleadoBaseMasComision4.ToString(),
32      "ingresos", empleadoBaseMasComision4.Ingresos() );
33
34      // invoca a ToString e Ingresos en objeto de clase derivada,
35      // usando variable de clase base
36      EmpleadoPorComision3 empleadoPorComision2 =
37          empleadoBaseMasComision4;
38      Console.WriteLine( "{0} {1}:\n{n{2}}\n{n{3}}: {4:C}",
39          "Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase",
40          "base a objeto de clase derivada",
41          empleadoPorComision2.ToString(), "ingresos",
42          empleadoPorComision2.Ingresos() );
43  } // fin de Main
44 } // fin de la clase PruebaPolimorfismo

```

Llama a ToString de EmpleadoPorComision3 con referencia de clase base a objeto de clase base:

```

empleado por comisión: Sue Jones
número de seguro social: 222-22-2222
ventas brutas: $10,000.00
tarifa de comisión: 0.06
ingresos: $600.00

```

Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase derivada a objeto de clase derivada:

```

salario base + empleado por comisión: Bob Lewis
número de seguro social: 333-33-3333
ventas brutas: $5,000.00
tarifa de comisión: 0.04
salario base: $300.00
ingresos: $500.00

```

Llama a ToString de EmpleadoBaseMasComision4 con referencia de clase base a objeto de clase derivada:

```

salario base + empleado por comisión: Bob Lewis
número de seguro social: 333-33-3333
ventas brutas: $5,000.00
tarifa de comisión: 0.04
salario base: $300.00
ingresos: $500.00

```

Figura 11.1 | Asignación de referencias a la clase base y a la clase derivada a variables de la clase base y de la clase derivada. (Parte 2 de 2).

En la figura 11.1, las líneas 11-12 crean un nuevo objeto `EmpleadoPorComision3` y asignan su referencia a una variable `EmpleadoPorComision3`. Las líneas 15-17 crean un nuevo objeto `EmpleadoBaseMasComision4` y asignan su referencia a una variable `EmpleadoBaseMasComision4`. Estas asignaciones son naturales; por ejemplo, el principal propósito de una variable `EmpleadoPorComision3` es guardar una referencia a un objeto `EmpleadoPorComision3`. Las líneas 21-24 utilizan la referencia `empleadoPorComision` para invocar a los métodos `ToString` e `Ingresos`. Como `empleadoPorComision` hace referencia a un objeto `EmpleadoPorComision3`, se hacen llamadas a los métodos de la versión de la clase base `EmpleadoPorComision3`. De manera similar, las líneas 28-32 utilizan a `empleadoBaseMasComision` para invocar a los métodos `ToString` e `Ingresos` en el objeto `EmpleadoBaseMasComision4`. Esto invoca a la versión de los métodos de la clase derivada `EmpleadoBaseMasComision4`.

Después, las líneas 36-37 asignan la referencia al objeto `empleadoBaseMasComision` de la clase derivada a una variable de la clase base `EmpleadoPorComision3`, que las líneas 38-42 utilizan para invocar a los métodos `ToString` e `Ingresos`. En sí, una variable de la clase base que contiene una referencia a un objeto de la clase derivada y se utiliza para llamar a un método `virtual`, es la que llama a la versión del método de la clase derivada que lo redefine. Por ende, `empleadoPorComision2.ToString()` en la línea 41 en realidad llama al método `ToString` de la clase base `EmpleadoBaseMasComision4`. El compilador permite este “cruzamiento”, ya que un objeto de una clase derivada *es un* objeto de su clase base (pero no viceversa). Cuando el compilador encuentra una llamada a un método que se realiza a través de una variable, determina si el método puede llamarse verificando el tipo de clase de la *variable*. Si esa clase contiene la declaración del método apropiada (o hereda uno), el compilador permite compilar la llamada. En tiempo de ejecución, *el tipo del objeto al cual se refiere la variable* es el que determina el método que se va a utilizar.

11.4 Clases y métodos abstractos

Cuando pensamos en un tipo de clase, asumimos que las aplicaciones crearán objetos de ese tipo. No obstante, en algunos casos es conveniente declarar clases para las cuales el programador nunca creará instancias de objetos. A dichas clases se les conoce como *clases abstractas*. Como se utilizan sólo como clases base en jerarquías de herencia, nos referimos a ellas como *clases base abstractas*. Estas clases no pueden utilizarse para instancias objetos, ya que como veremos pronto, las clases abstractas están incompletas; las clases derivadas deben declarar las “piezas faltantes”. En la sección 11.5.1 demostraremos las clases abstractas.

El propósito principal de una clase abstracta es proporcionar una clase base apropiada, a partir de la cual puedan heredar otras clases y, por ende, compartir un diseño común. Por ejemplo, en la jerarquía de `Figura` de la figura 10.3, las clases derivadas heredan la noción de lo que significa ser una `Figura`; los atributos comunes tales como `posicion`, `color` y `grosorBorde`, y los comportamientos tales como `Dibujar`, `Mover`, `CambiarTamanio` y `CambiarColor`. Las clases que pueden utilizarse para instanciar objetos se llaman *clases concretas*. Dichas clases proporcionan implementaciones de *cada* método que declaran (algunas de las implementaciones pueden heredarse). Por ejemplo, podríamos derivar las clases concretas `Circulo`, `Cuadrado` y `Triangulo` de la clase base abstracta `FiguraBidimensional`. De manera similar, podríamos derivar las clases concretas `Esfera`, `Cubo` y `Tetraedro` de la clase base abstracta `FiguraTridimensional`. Las clases base abstractas son demasiado generales como para crear objetos reales; sólo especifican lo que tienen en común las clases derivadas. Necesitamos ser más específicos para poder crear objetos. Por ejemplo, si envía el mensaje `Dibujar` a la clase abstracta `FiguraBidimensional`, la clase sabe que las figuras bidimensionales deben poder dibujarse, pero no sabe qué figura específica dibujar, por lo que no puede implementar un verdadero método `Dibujar`. Las clases concretas proporcionan los detalles específicos que hacen razonable la creación de instancias de objetos.

No todas las jerarquías de herencia contienen clases abstractas. Sin embargo, a menudo los programadores escriben código cliente que utiliza sólo tipos de clases bases abstractas para reducir las dependencias del código cliente en un rango de tipos de clases derivadas específicas. Por ejemplo, un programador puede escribir un método con un parámetro de un tipo de clase base abstracta. Cuando se llama, ese método puede recibir un objeto de cualquier clase concreta que extienda en forma directa o indirecta a la clase base especificada como el tipo del parámetro.

Algunas veces las clases abstractas constituyen varios niveles de la jerarquía. Por ejemplo, la jerarquía de `Figura` de la figura 10.3 empieza con la clase abstracta `Figura`. En el siguiente nivel de la jerarquía hay dos clases abstractas más, `FiguraBidimensional` y `FiguraTridimensional`. El siguiente nivel de la jerarquía declara clases concretas para objetos de `FiguraBidimensional` (`Circulo`, `Cuadrado` y `Triangulo`) y para objetos de `FiguraTridimensional` (`Esfera`, `Cubo` y `Tetraedro`).

Para hacer una clase abstracta, ésta se declara con la palabra clave `abstract`. Por lo general, una clase base abstracta contiene uno o más *métodos abstractos*. Un método abstracto tiene la palabra clave `abstract` en su declaración, como en

```
public abstract void Dibujar(); // método abstracto
```

Los métodos abstractos no proporcionan implementaciones. Una clase que contiene métodos abstractos debe declararse como clase abstracta, aún si esa clase contiene métodos concretos (no abstractos). Cada clase derivada concreta de una clase base abstracta también debe proporcionar implementaciones concretas de los métodos

abstractos de la clase base. En la figura 11.4 mostramos un ejemplo de una clase abstracta con un método abstracto.

Las propiedades también pueden declararse como **abstract** y después se redefinen en clases derivadas con la palabra clave **override**, justo igual que los métodos. Esto permite a una clase base abstracta especificar propiedades comunes de sus clases derivadas. Las declaraciones de propiedades abstractas tienen la siguiente forma:

```
public abstract TipoPropiedad MiPropiedad
{
    get;
    set;
} // fin de la propiedad abstracta
```

Los signos de punto y coma después de las palabras clave **get** y **set** indican que no se proporciona la implementación para estos descriptores de acceso. Una propiedad abstracta puede omitir las implementaciones para el descriptor de acceso **get**, el descriptor de acceso **set** o ambos. Las clases derivadas concretas deben proporcionar implementaciones para *cada* descriptor de acceso declarado en la propiedad abstracta. Cuando se especifican ambos descriptores de acceso **get** y **set** (como en la declaración anterior), cada clase derivada concreta debe implementar ambos descriptores. Si se omite un descriptor de acceso, no se permite que la clase derivada lo implemente. Esto produce un error de compilación.

Los constructores y los métodos **static** no pueden declararse **abstract**. Los constructores no se heredan, por lo que un constructor **abstract** nunca podría implementarse. De manera similar, las clases derivadas no pueden redefinir métodos **static**, por lo que un método **abstract static** nunca podría implementarse.



Observación de ingeniería de software 11.2

Una clase abstracta declara los atributos y comportamientos comunes de las diversas clases que heredan de ella, ya sea en forma directa o indirecta, en una jerarquía de clases. Por lo general, una clase abstracta contiene uno o más métodos o propiedades abstractos, que las clases derivadas concretas deben redefinir. Las variables de instancia, los métodos y las propiedades concretas de una clase abstracta están sujetos a las reglas normales de la herencia.



Error común de programación 11.1

Tratar de instanciar un objeto de una clase abstracta es un error de compilación.



Error común de programación 11.2

Si no se implementan los métodos y las propiedades abstractas de la clase base en una clase derivada, se produce un error de compilación, a menos que la clase derivada también se declare como **abstract**.

Aunque no podemos instanciar objetos de clases base abstractas, pronto veremos que *podemos* usar clases base abstractas para declarar variables que puedan guardar referencias a objetos de cualquier clase concreta que se derive de esas clases abstractas. Por lo general, las aplicaciones utilizan dichas variables para manipular los objetos de las clases derivadas mediante el polimorfismo. Además, puede usar los nombres de las clases base abstractas para invocar métodos **static** que estén declarados en esas clases base abstractas.

En especial, el polimorfismo es efectivo para implementar los denominados sistemas de software en capas. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico puede operar en forma muy distinta a los demás. Aún así, los comandos comunes pueden leer o escribir datos desde y hacia los dispositivos. Para cada dispositivo, el sistema operativo utiliza una pieza de software llamada controlador de dispositivos para controlar toda la comunicación entre el sistema y el dispositivo. El mensaje de escritura que se envía a un objeto controlador de dispositivo necesita implementarse de manera específica en el contexto de ese controlador y la forma en que manipula a un dispositivo específico. No obstante, la llamada de escritura en sí no es distinta a la escritura en cualquier otro dispositivo en el sistema: colocar cierto número de bytes de memoria en ese dispositivo. Un sistema operativo orientado a objetos podría usar una clase base abstracta para proporcionar una “interfaz” apropiada para todos los controladores de dispositivos. Después, a través de la herencia de esa clase base abstracta, se forman clases derivadas que se comporten todas de manera similar. Los métodos del controlador de dispositivos se declaran como métodos abstractos en la clase base abstracta. Las implementaciones de estos métodos abstractos se proporcionan en las clases derivadas que corresponden a los tipos específicos de controladores de dispositivos. Siempre se

están desarrollando nuevos dispositivos, a menudo mucho después de que se ha liberado el sistema operativo. Cuando usted compra un nuevo dispositivo, éste incluye un controlador de dispositivo proporcionado por el distribuidor. El dispositivo opera de inmediato, una vez que usted lo conecta a la computadora e instala el controlador de dispositivo. Éste es otro elegante ejemplo acerca de cómo el polimorfismo hace que los sistemas sean extensibles.

En la programación orientada a objetos es común declarar una *clase iteradora* que pueda recorrer todos los objetos en una colección, como un arreglo (capítulo 8) o un arreglo *ArrayList* (capítulo 26, Colecciones). Por ejemplo, una aplicación puede imprimir un arreglo *ArrayList* de objetos creando un objeto iterador, y luego usándolo para obtener el siguiente elemento de la lista cada vez que se llame al iterador. Los iteradores se utilizan comúnmente en la programación polimórfica para recorrer una colección que contiene referencias a objetos de diversas clases en una jerarquía de herencia. (Los capítulos 25-26 presentan un tratamiento detallado de las nuevas capacidades “genéricas”, *ArrayList* y los iteradores). Por ejemplo, un arreglo *ArrayList* de referencias a objetos de la clase *FiguraBidimensional* podría contener referencias a objetos de las clases derivadas *Cuadrado*, *Círculo*, *Triangulo* y así, sucesivamente. Si utilizáramos el polimorfismo para llamar al método *Dibujar* para cada objeto *FiguraBidimensional* mediante una variable *FiguraBidimensional*, se dibujaría a cada objeto correctamente en la pantalla.

11.5 Caso de estudio: sistema de nómina utilizando polimorfismo

En esta sección analizamos de nuevo la jerarquía *EmpleadoPorComision-EmpleadoBaseMasComision* que exploramos a lo largo de la sección 10.4. Ahora podemos usar un método abstracto y polimorfismo para realizar cálculos de nómina, con base en el tipo de empleado. Vamos a crear una jerarquía de empleados mejorada para resolver el siguiente problema:

Una compañía paga a sus empleados por semana. Los empleados son de cuatro tipos: empleados asalariados que reciben un salario semanal fijo, sin importar el número de horas trabajadas; empleados por horas, que reciben un sueldo por hora y pago por tiempo extra, por todas las horas trabajadas que excedan a 40 horas; empleados por comisión, que reciben un porcentaje de sus ventas y empleados asalariados por comisión, que reciben un salario base más un porcentaje de sus ventas. Para este periodo de pago, la compañía ha decidido recompensar a los empleados asalariados por comisión, agregando un 10% a sus salarios base. La compañía desea implementar una aplicación en C# que realice sus cálculos de nómina en forma polimórfica.

Utilizaremos la clase abstracta *Empleado* para representar el concepto general de un empleado. Las clases que extienden a *Empleado* son *EmpleadoAsalariado*, *EmpleadoPorComision* y *EmpleadoPorHoras*. La clase *EmpleadoBaseMasComision* (que extiende a *EmpleadoPorComision*) representa el último tipo de empleado. El diagrama de clases de UML en la figura 11.2 muestra la jerarquía de herencia para nuestra aplicación polimórfica de nómina de empleados. Observe que la clase abstracta *Empleado* está en cursivas, según la convención de UML.

La clase base abstracta *Empleado* declara la “interfaz” para la jerarquía; esto es, el conjunto de métodos que puede invocar una aplicación en todos los objetos *Empleado*. Aquí utilizamos el término “interfaz” en un sentido general, para referirnos a las diversas formas en que las aplicaciones pueden comunicarse con los objetos de cualquier clase derivada de *Empleado*. Tenga cuidado de no confundir la noción general de una “interfaz” con la noción formal de una interfaz en C#, el tema de la sección 11.7. Cada empleado, sin importar la manera en que se calculen sus ingresos, tiene un primer nombre, un apellido paterno y un número de seguro social, por lo que las variables de instancia private *primerNombre*, *apellidoPaterno* y *numeroSeguroSocial* aparecen en la clase base abstracta *Empleado*.



Observación de ingeniería de software 11.3

Una clase derivada puede heredar la “interfaz” o “implementación” de una clase base. Las jerarquías diseñadas para la herencia de implementación tienden a tener su funcionalidad en niveles altos de la jerarquía; cada nueva clase derivada hereda uno o más métodos que se implementaron en una clase base, y la clase derivada utiliza las implementaciones de la clase base. Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad en niveles bajos de la jerarquía; una clase base especifica uno o más métodos abstractos que deben declararse para cada clase concreta en la jerarquía, y las clases derivadas individuales redifinen estos métodos para proporcionar la implementación específica para cada clase derivada.

Las siguientes secciones implementan la jerarquía de clases de `Empleado`. La primera sección implementa la clase base `abstract Empleado`. Las siguientes cuatro secciones implementan cada una de las clases concretas. La sexta sección implementa una aplicación de prueba que crea objetos de todas estas clases y procesa esos objetos mediante el polimorfismo.

11.5.1 Creación de la clase base abstracta `Empleado`

La clase `Empleado` (figura 11.4) proporciona los métodos `Ingresos` y `ToString`, además de las propiedades que manipulan las variables de instancia de `Empleado`. Es evidente que un método `Ingresos` se aplica en forma general a todos los empleados. Pero cada cálculo de los ingresos depende de la clase de empleado. Por lo tanto, declaramos a `Ingresos` como `abstract` en la clase base `Empleado`, ya que una implementación predeterminada no tiene sentido para ese método; no hay suficiente información para determinar qué monto debe devolver `Ingresos`. Cada una de las clases derivadas redefine a `Ingresos` con una implementación apropiada. Para calcular los ingresos de un empleado, la aplicación asigna una referencia al objeto de empleado a una variable de la clase base `Empleado`, y después invoca al método `Ingresos` en esa variable. Mantenemos un arreglo de variables `Empleado`, cada una de las cuales guarda una referencia a un objeto `Empleado` (desde luego que no puede haber objetos `Empleado`, ya que ésta es una clase abstracta; sin embargo, debido a la herencia todos los objetos de todas las clases derivadas de `Empleado` pueden considerarse como objetos `Empleado`). La aplicación itera a través del arreglo y llama al método `Ingresos` para cada objeto `Empleado`. C# procesa estas llamadas a los métodos en forma polimórfica. Al incluir a `Ingresos` como un método abstracto en `Empleado`, se obliga a cada clase concreta derivada de `Empleado` a redefinir `Ingresos` con un método que realice el cálculo apropiado del salario.

El método `ToString` en la clase `Empleado` devuelve un objeto `string` que contiene el primer nombre, el apellido paterno y el número de seguro social del empleado. Cada clase derivada de `Empleado` redefine al método `ToString` para crear una representación `string` de un objeto de esa clase que contiene el tipo del empleado (por ejemplo, "empleado asalariado:"), seguido del resto de la información del empleado.

El diagrama en la figura 11.3 muestra cada una de las cinco clases en la jerarquía, hacia abajo en la columna de la izquierda, y los métodos `Ingresos` y `ToString` en la fila superior. Para cada clase, el diagrama muestra los resultados deseados de cada método. [Nota: no listamos las propiedades de la clase base `Empleado` porque no se redefinen en ninguna de las clases derivadas; cada una de estas propiedades se hereda y cada una de las clases derivadas las utiliza "como están".]

Consideremos ahora la declaración de la clase `Empleado` (figura 11.4). Esta clase incluye un constructor que recibe el primer nombre, el apellido paterno y el número de seguro social como argumentos (líneas 10-15); las propiedades de sólo lectura para obtener el primer nombre, el apellido paterno y el número de seguro social (líneas 18-24, 27-33 y 36-42, respectivamente); el método `ToString` (líneas 45-49), el cual utiliza las propiedades para devolver la representación `string` de `Empleado`; y el método `abstract Ingresos` (línea 52), que las clases derivadas concretas deben implementar. Observe que el constructor de `Empleado` no valida el número de seguro social en este ejemplo. Por lo general, se debe proporcionar esa validación.

¿Por qué declaramos a `Ingresos` como un método abstracto? Simplemente, no tiene sentido proporcionar una implementación de este método en la clase `Empleado`. No podemos calcular los ingresos para un `Empleado`

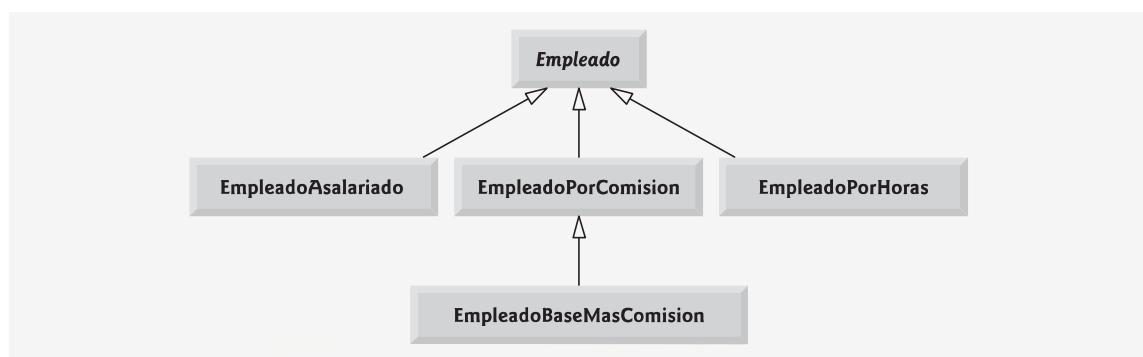


Figura 11.2 | Diagrama de clases de UML para la jerarquía de `Empleado`.

	Ingresos	ToString
Empleado	abstract	primerNombre apellidoPaterno número de seguro social: NSS
Empleado-Asalariado	salarioSemanal	empleado asalariado: primerNombre apellidoPaterno número de seguro social: NSS salario semanal: salarioSemanal
Empleado-PorHoras	<i>Si horas <= 40</i> <i>sueLdo * horas</i> <i>Si horas > 40</i> <i>40 * sueldo +</i> <i>(horas - 40) * sueldo * 1.5</i>	empleado por horas: primerNombre apellidoPaterno número de seguro social: NSS sueldo por horas: sueldo horas trabajadas: horas
Empleado PorComision	tarifaComision * ventasBrutas	empleado por comisión: primerNombre apellidoPaterno número de seguro social: NSS ventas brutas: ventasBrutas tarifa de comisión: tarifaComision
Empleado-BaseMas-Comision	(tarifaComision * ventasBrutas) + salarioBase	empleado por comisión con salario base: primerNombre apellidoPaterno número de seguro social: NSS ventas brutas: ventasBrutas tarifa de comisión: tarifaComision salario base: salarioBase

Figura 11.3 | Interfaz polimórfica para las clases de la jerarquía de Empleado.

general; primero debemos conocer el tipo de Empleado específico para determinar el cálculo apropiado de los ingresos. Al declarar este método **abstract**, indicamos que cada clase derivada concreta *debe* proporcionar una implementación apropiada para **Ingresos**, y que una aplicación podrá utilizar las variables de la clase base **Empleado** para invocar al método **Ingresos** en forma polimórfica, para cualquier tipo de **Empleado**.

```

1 // Fig. 11.4: Empleado.cs
2 // La clase base abstracta Empleado.
3 public abstract class Empleado
4 {
5     private string primerNombre;
6     private string apellidoPaterno;
7     private string numeroSeguroSocial;
8
9     // constructor con tres parámetros
10    public Empleado( string nombre, string apellido, string nss )
11    {
12        primerNombre = nombre;
13        apellidoPaterno = apellido;
14        numeroSeguroSocial = nss;
15    } // fin de constructor de Empleado con tres parámetros
16
17    // propiedad de sólo lectura que obtiene el primer nombre del empleado
18    public string PrimerNombre
19    {
```

Figura 11.4 | La clase base abstracta Empleado. (Parte 1 de 2).

```

20     get
21     {
22         return primerNombre;
23     } // fin de get
24 } // fin de la propiedad PrimerNombre
25
26 // propiedad de sólo lectura que obtiene el apellido paterno del empleado
27 public string ApellidoPaterno
28 {
29     get
30     {
31         return apellidoPaterno;
32     } // fin de get
33 } // fin de la propiedad ApellidoPaterno
34
35 // propiedad de sólo lectura que obtiene el número de seguro social del empleado
36 public string NumeroSeguroSocial
37 {
38     get
39     {
40         return numeroSeguroSocial;
41     } // fin de get
42 } // fin de la propiedad NumeroSeguroSocial
43
44 // devuelve representación string del objeto Empleado, usando las propiedades
45 public override string ToString()
46 {
47     return string.Format( "{0} {1}\nNúmero de seguro social: {2}",
48     PrimerNombre, ApellidoPaterno, NumeroSeguroSocial );
49 } // fin del método ToString
50
51 // método abstracto redefinido por las clases derivadas
52 public abstract decimal Ingresos(); // no hay implementación aquí
53 } // fin de la clase abstracta Empleado

```

Figura 11.4 | La clase base abstracta Empleado. (Parte 2 de 2).

11.5.2 Creación de la clase derivada concreta EmpleadoAsalariado

La clase EmpleadoAsalariado (figura 11.5) extiende a la clase Empleado (línea 3) y redefine a Ingresos (líneas 28-31), lo cual convierte a EmpleadoAsalariado en una clase concreta. La clase incluye un constructor (líneas 8-12) que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal como argumentos; la propiedad SalarioSemanal para manipular la variable de instancia salarioSemanal, incluyendo un descriptor de acceso set que asegura que asignemos sólo valores no negativos a salarioSemanal (líneas 15-25); el método Ingresos (líneas 28-31) para calcular los ingresos de un EmpleadoAsalariado; y el método ToString (líneas 34-38) que devuelve un objeto string que incluye el tipo del empleado, a saber, "empleado asalariado:", seguido de la información específica para el empleado producida por el método ToString de la clase base Empleado y la propiedad SalarioSemanal de EmpleadoAsalariado. El constructor de la clase EmpleadoAsalariado pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de Empleado (línea 9) a través de un inicializador de constructor, para inicializar las variables de instancia private que no se heredan de la clase base. El método Ingresos redefine al método abstracto Ingresos de Empleado para proporcionar una implementación concreta que devuelva el salario semanal del EmpleadoAsalariado. Si no implementamos Ingresos, la clase EmpleadoAsalariado debe declararse como abstract; en caso contrario, se produce un error de compilación (y desde luego, queremos que EmpleadoAsalariado sea una clase concreta).

El método ToString (líneas 34-38) de la clase EmpleadoAsalariado redefine al método ToString de Empleado. Si la clase EmpleadoAsalariado no redefiniera a ToString, EmpleadoAsalariado habría heredado la versión de ToString de Empleado. En ese caso, el método ToString de EmpleadoAsalariado simplemente

```

1 // Fig. 11.5: EmpleadoAsalariado.cs
2 // La clase EmpleadoAsalariado que extiende a Empleado.
3 public class EmpleadoAsalariado : Empleado
4 {
5     private decimal salarioSemanal;
6
7     // constructor con cuatro parámetros
8     public EmpleadoAsalariado( string nombre, string apellido, string nss,
9         decimal salario ) : base( nombre, apellido, nss )
10    {
11        SalarioSemanal = salario; // valida el salario a través de una propiedad
12    } // fin del constructor de EmpleadoAsalariado con cuatro parámetros
13
14    // propiedad que obtiene y establece el salario del empleado asalariado
15    public decimal SalarioSemanal
16    {
17        get
18        {
19            return SalarioSemanal;
20        } // fin de get
21        set
22        {
23            SalarioSemanal = ( ( value >= 0 ) ? value : 0 ); // validación
24        } // fin de set
25    } // fin de la propiedad SalarioSemanal
26
27    // calcula los ingresos; redefine el método abstracto Ingresos en Empleado
28    public override decimal Ingresos()
29    {
30        return SalarioSemanal;
31    } // fin del método Ingresos
32
33    // devuelve representación string del objeto EmpleadoAsalariado
34    public override string ToString()
35    {
36        return string.Format( "empleado asalariado: {0}\n{1}: {2:C}",
37            base.ToString(), "salario semanal", SalarioSemanal );
38    } // fin del método ToString
39 } // fin de la clase EmpleadoAsalariado

```

Figura 11.5 | La clase EmpleadoAsalariado que extiende a Empleado.

devolvería el nombre completo del empleado y su número de seguro social, lo cual no representa en forma adecuada a un EmpleadoAsalariado. Para producir una representación *string* completa de EmpleadoAsalariado, el método *ToString* de la clase derivada devuelve "empleado asalariado: ", seguido de la información específica de la clase base Empleado (es decir, el primer nombre, el apellido paterno y el número de seguro social) que se obtiene al invocar el método *ToString* de la clase base (línea 37); éste es un excelente ejemplo de reutilización de código. La representación *string* de un EmpleadoAsalariado también contiene el salario semanal del empleado, el cual se obtiene mediante el uso de la propiedad *SalarioSemanal* de la clase.

11.5.3 Creación de la clase derivada concreta EmpleadoPorHoras

La clase EmpleadoPorHoras (figura 11.6) también extiende a la clase Empleado (línea 3). La clase incluye un constructor (líneas 9-15) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un sueldo por horas y el número de horas trabajadas. Las líneas 18-28 y 31-42 declaran las propiedades *Sueldo* y *Horas* para las variables de instancia *sueldo* y *horas*, respectivamente. El descriptor de acceso *set* en la propiedad *Sueldo* (líneas 24-27) asegura que *sueldo* sea no negativo, y el descriptor de acceso *set* en la propiedad *Horas* (líneas 37-41) asegura que *horas* se encuentre en el rango de 0-168 (el número total de horas

en la semana). La clase `EmpleadoPorHoras` también incluye el método `Ingresos` (líneas 45-51) para calcular los ingresos de un `EmpleadoPorHoras`; y el método `ToString` (líneas 54-59), que devuelve el tipo del empleado, a saber, "empleado por horas:", e información específica para ese empleado. Observe que el constructor de `EmpleadoPorHoras`, al igual que el constructor de `EmpleadoAsalariado`, pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de la clase base `Empleado` (línea 11) para inicializar las variables de instancia `private` de la clase base. Además, el método `ToString` llama al método `ToString` de la clase base (línea 58) para obtener la información específica del `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social); éste es otro excelente ejemplo de reutilización de código.

```

1 // Fig. 11.6: EmpleadoPorHoras.cs
2 // La clase EmpleadoPorHoras que extiende a Empleado.
3 public class EmpleadoPorHoras : Empleado
4 {
5     private decimal sueldo; // sueldo por hora
6     private decimal horas; // horas trabajadas durante la semana
7
8     // constructor con cinco parámetros
9     public EmpleadoPorHoras( string nombre, string apellido, string nss,
10         decimal sueldoPorHoras, decimal horasTrabajadas )
11         : base( nombre, apellido, nss )
12     {
13         Sueldo = sueldoPorHoras; // valida el sueldo por horas a través de una propiedad
14         Horas = horasTrabajadas; // valida las horas trabajadas a través de una propiedad
15     } // fin del constructor de EmpleadoPorHoras con cinco parámetros
16
17     // propiedad que obtiene y establece el sueldo del empleado por horas
18     public decimal Sueldo
19     {
20         get
21         {
22             return sueldo;
23         } // fin de get
24         set
25         {
26             sueldo = ( value >= 0 ) ? value : 0; // validación
27         } // fin de set
28     } // fin de la propiedad Sueldo
29
30     // propiedad que obtiene y establece las horas del empleado por horas
31     public decimal Horas
32     {
33         get
34         {
35             return horas;
36         } // fin de get
37         set
38         {
39             horas = ( ( value >= 0 ) && ( value <= 168 ) ) ?
40                     value : 0; // validación
41         } // fin de set
42     } // fin de la propiedad Horas
43
44     // calcula los ingresos; redefine el método abstracto Ingresos de Empleado
45     public override decimal Ingresos()
46     {
47         if ( Horas <= 40 ) // no hay tiempo extra

```

Figura 11.6 | La clase `EmpleadoPorHoras` que extiende a `Empleado`. (Parte I de 2).

```

48     return Sueldo * Horas;
49   else
50     return ( 40 * Sueldo ) + ( ( Horas - 40 ) * Sueldo * 1.5M );
51 } // fin del método Ingresos
52
53 // devuelve representación string del objeto EmpleadoPorHoras
54 public override string ToString()
55 {
56   return string.Format(
57     "empleado por horas: {0}\n{1}: {2:C}; {3}: {4:F2}",
58     base.ToString(), "sueldo por horas", Sueldo, "horas trabajadas", Horas );
59 } // fin del método ToString
60 } // fin de la clase EmpleadoPorHoras

```

Figura 11.6 | La clase EmpleadoPorHoras que extiende a Empleado. (Parte 2 de 2).

11.5.4 Creación de la clase derivada concreta EmpleadoPorComision

La clase EmpleadoPorComision (figura 11.7) extiende a la clase Empleado (línea 3). Esta clase incluye a un constructor (líneas 9-14) que recibe como argumentos un primer nombre, un apellido, un número de seguro social, un monto de ventas y una tarifa de comisión; las propiedades (líneas 17-28 y 31-41) para las variables de instancia tarifaComision y ventasBrutas, respectivamente; el método Ingresos (líneas 44-47) para calcular los ingresos de un EmpleadoPorComision; y el método ToString (líneas 50-55) que devuelve el tipo del empleado, a saber, "empleado por comisión: ", e información específica del empleado.

```

1 // Fig. 11.7: EmpleadoPorComision.cs
2 // La clase EmpleadoPorComision que extiende a Empleado.
3 public class EmpleadoPorComision : Empleado
4 {
5   private decimal ventasBrutas; // ventas semanales totales
6   private decimal tarifaComision; // porcentaje de comisión
7
8   // constructor con cinco parámetros
9   public EmpleadoPorComision( string nombre, string apellido, string nss,
10     decimal ventas, decimal tarifa ) : base( nombre, apellido, nss )
11   {
12     VentasBrutas = ventas; // valida las ventas brutas a través de una propiedad
13     TarifaComision = tarifa; // valida la tarifa de comisión a través de una propiedad
14 } // fin del constructor de EmpleadoPorComision con cinco parámetros
15
16 // propiedad que obtiene y establece la tarifa de comisión del empleado por comisión
17 public decimal TarifaComision
18 {
19   get
20   {
21     return tarifaComision;
22   } // fin de get
23   set
24   {
25     tarifaComision = ( value > 0 && value < 1 ) ?
26       value : 0; // validación
27   } // fin de set
28 } // fin de la propiedad TarifaComision
29

```

Figura 11.7 | La clase EmpleadoPorHoras que extiende a Empleado. (Parte 1 de 2).

```

30  // propiedad que obtiene y establece las ventas brutas del empleado por comisión
31  public decimal VentasBrutas
32  {
33      get
34      {
35          return VentasBrutas;
36      } // fin de get
37      set
38      {
39          VentasBrutas = ( value >= 0 ) ? value : 0; // validación
40      } // fin de set
41  } // fin de la propiedad VentasBrutas
42
43  // calcula los ingresos; redefine el método abstracto Ingresos en Empleado
44  public override decimal Ingresos()
45  {
46      return TarifaComision * VentasBrutas;
47  } // fin del método Ingresos
48
49  // devuelve representación string del objeto EmpleadoPorComision
50  public override string ToString()
51  {
52      return string.Format( "{0}: {1}\n{2}: {3:C}\n{4}: {5:F2}",
53          "empleado por comisión", base.ToString(),
54          "ventas brutas", VentasBrutas, "tarifa de comisión", TarifaComision );
55  } // fin del método ToString
56 } // fin de la clase EmpleadoPorComision

```

Figura 11.7 | La clase EmpleadoPorHoras que extiende a Empleado. (Parte 2 de 2).

El constructor de EmpleadoPorComision también pasa el primer nombre, el apellido y el número de seguro social al constructor de Empleado (línea 10) para inicializar las variables de instancia `private` de Empleado. El método `ToString` llama al método `ToString` de la clase base (línea 53) para obtener la información específica del Empleado (es decir, primer nombre, apellido paterno y número de seguro social).

11.5.5 Creación de la clase derivada concreta indirecta EmpleadoBaseMasComision

La clase EmpleadoBaseMasComision (figura 11.8) extiende a la clase EmpleadoPorComision (línea 3) y, por lo tanto, es una clase derivada indirecta de la clase Empleado. La clase EmpleadoBaseMasComision tiene un constructor (líneas 8-13) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas, una tarifa de comisión y un salario base. Después pasa el primer nombre, el apellido paterno, el número de seguro social, el monto de ventas y la tarifa de comisión al constructor de EmpleadoPorComision (línea 10) para inicializar los miembros de datos `private` de la clase base. EmpleadoBaseMasComision también contiene la propiedad `SalarioBase` (líneas 17-27) para manipular la variable de instancia `salarioBase`. El método `Ingresos` (líneas 30-33) calcula los ingresos de un EmpleadoBaseMasComision. Observe que la línea 32 en el método `Ingresos` llama al método `Ingresos` de la clase base EmpleadoPorComision para calcular la porción basada en la comisión de los ingresos del empleado. Éste es otro buen ejemplo de reutilización de código. El método `ToString` de EmpleadoBaseMasComision (líneas 36-40) crea una representación `string` de un EmpleadoBaseMasComision, la cual contiene "salario base +", seguida del objeto `string` que se obtiene al invocar el método `ToString` de la clase base EmpleadoPorComision (otro buen ejemplo de reutilización de código), y después el salario base. El resultado es un objeto `string` que empieza con "salario base + empleado por comisión", seguido del resto de la información de EmpleadoBaseMasComision. Recuerde que el método `ToString` de EmpleadoPorComision obtiene el primer nombre, el apellido paterno y el número de seguro social del empleado mediante la invocación del método `ToString` de su clase base (es decir, Empleado); otro ejemplo

```

1 // Fig. 11.8: EmpleadoBaseMasComision.cs
2 // La clase EmpleadoBaseMasComision que extiende a EmpleadoPorComision.
3 public class EmpleadoBaseMasComision : EmpleadoPorComision
4 {
5     private decimal salarioBase; // salario base por semana
6
7     // constructor con seis parámetros
8     public EmpleadoBaseMasComision( string nombre, string apellido,
9         string nss, decimal ventas, decimal tarifa, decimal salario )
10        : base( nombre, apellido, nss, ventas, tarifa )
11    {
12        SalarioBase = salario; // valida el salario base a través de una propiedad
13    } // fin del constructor de EmpleadoBaseMasComision con seis parámetros
14
15    // propiedad que obtiene y establece
16    // el salario base del empleado por comisión con salario base
17    public decimal SalarioBase
18    {
19        get
20        {
21            return salarioBase;
22        } // fin de get
23        set
24        {
25            salarioBase = ( value >= 0 ) ? value : 0; // validación
26        } // fin de set
27    } // fin de la propiedad SalarioBase
28
29    // calcula los ingresos; redefine el método Ingresos en EmpleadoPorComision
30    public override decimal Ingresos()
31    {
32        return SalarioBase + base.Ingresos();
33    } // fin del método Ingresos
34
35    // devuelve representación string del objeto EmpleadoBaseMasComision
36    public override string ToString()
37    {
38        return string.Format( "{0} {1}; {2}: {3:C}",
39            "salario base +", base.ToString(), "salario base", SalarioBase );
40    } // fin del método ToString
41 } // fin de la clase EmpleadoBaseMasComision

```

Figura 11.8 | La clase EmpleadoBaseMasComision que extiende a EmpleadoPorComision.

más de reutilización de código. Observe que el método `ToString` de `EmpleadoBaseMasComision` inicia una cadena de llamadas a métodos que abarcan los tres niveles de la jerarquía de `Empleado`.

11.5.6 El procesamiento polimórfico, el operador `is` y la conversión descendente

Para probar nuestra jerarquía de `Empleado`, la aplicación en la figura 11.9 crea un objeto de cada una de las cuatro clases concretas `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`. La aplicación manipula estos objetos, primero mediante variables del mismo tipo de cada objeto y después mediante el polimorfismo, utilizando un arreglo de variables `Empleado`. Al procesar los objetos mediante el polimorfismo, la aplicación incrementa el salario base de cada `EmpleadoBaseMasComision` en 10% (desde luego que para esto se requiere determinar el tipo del objeto en tiempo de ejecución). Por último, la aplicación determina e imprime en forma polimórfica el tipo de cada objeto en el arreglo `Empleado`. Las líneas 10-20 crean objetos de cada una de las cuatro clases concretas derivadas de `Empleado`. Las líneas 24-32 imprimen en pantalla

la representación `string` y los ingresos de cada uno de estos objetos. Observe que `Write` llama en forma implícita al método `ToString` de cada objeto, cuando éste se imprime en pantalla como un objeto `string` con elementos de formato.

```

1 // Fig. 11.9: PruebaSistemaNomina.cs
2 // Aplicación de prueba de la jerarquía de Empleado.
3 using System;
4
5 public class PruebaSistemaNomina
6 {
7     public static void Main( string[] args )
8     {
9         // crea objetos de clases derivadas
10        EmpleadoAsalariado empleadoAsalariado =
11            new EmpleadoAsalariado( "John", "Smith", "111-11-1111", 800.00M );
12        EmpleadoPorHoras empleadoPorHoras =
13            new EmpleadoPorHoras( "Karen", "Price",
14                "222-22-2222", 16.75M, 40.0M );
15        EmpleadoPorComision empleadoPorComision =
16            new EmpleadoPorComision( "Sue", "Jones",
17                "333-33-3333", 10000.00M, .06M );
18        EmpleadoBaseMasComision empleadoBaseMasComision =
19            new EmpleadoBaseMasComision( "Bob", "Lewis",
20                "444-44-4444", 5000.00M, .04M, 300.00M );
21
22        Console.WriteLine( "Empleados procesados en forma individual:\n" );
23
24        Console.WriteLine( "{0}\n{1}: {2:C}\n",
25            empleadoAsalariado, "ingresos", empleadoAsalariado.Ingresos() );
26        Console.WriteLine( "{0}\n{1}: {2:C}\n",
27            empleadoPorHoras, "ingresos", empleadoPorHoras.Ingresos() );
28        Console.WriteLine( "{0}\n{1}: {2:C}\n",
29            empleadoPorComision, "ingresos", empleadoPorComision.Ingresos() );
30        Console.WriteLine( "{0}\n{1}: {2:C}\n",
31            empleadoBaseMasComision,
32            "ingresos", empleadoBaseMasComision.Ingresos() );
33
34        // crea arreglo Empleado de cuatro elementos
35        Empleado[] empleados = new Empleado[ 4 ];
36
37        // inicializa arreglo con objetos Empleado de tipos derivados
38        empleados[ 0 ] = empleadoAsalariado;
39        empleados[ 1 ] = empleadoPorHoras;
40        empleados[ 2 ] = empleadoPorComision;
41        empleados[ 3 ] = empleadoBaseMasComision;
42
43        Console.WriteLine( "Empleados procesados en forma polimórfica:\n" );
44
45        // procesa en forma generica cada elemento en el arreglo de empleados
46        foreach ( Empleado empleadoActual in empleados )
47        {
48            Console.WriteLine( empleadoActual ); // invoca a ToString
49
50            // determina si el elemento es un EmpleadoBaseMasComision
51            if ( empleadoActual is EmpleadoBaseMasComision )
52            {

```

Figura 11.9 | Aplicación de prueba de la jerarquía de Empleado. (Parte I de 3).

```

53         // conversión descendente de referencia de Empleado a
54         // referencia de EmpleadoBaseMasComision
55         EmpleadoBaseMasComision empleado =
56             ( EmpleadoBaseMasComision ) empleadoActual;
57
58         empleado.SalarioBase *= 1.10M;
59         Console.WriteLine(
60             "nuevo salario base con incremento del 10%: {0:C}",
61             empleado.SalarioBase );
62     } // fin de if
63
64     Console.WriteLine(
65         "ingresos {0:C}\n", empleadoActual.Ingresos() );
66 } // fin de foreach
67
68 // obtiene el nombre del tipo de cada objeto en el arreglo de empleados
69 for ( int j = 0; j < empleados.Length; j++ )
70     Console.WriteLine( "Empleado {0} es un {1}", j,
71         empleados[ j ].GetType() );
72 } // fin de Main
73 } // fin de la clase PruebaSistemaNomina

```

Empleados procesados en forma individual:

empleado asalariado: John Smith
 número de seguro social: 111-11-1111
 salario semanal: \$800.00
 ingresos: \$800.00

empleado por horas: Karen Price
 número de seguro social: 222-22-2222
 sueldo por horas: \$16.75; horas trabajadas: 40.00
 ingresos: \$670.00

empleado por comisión: Sue Jones
 número de seguro social: 333-33-3333
 ventas brutas: \$10,000.00
 tarifa de comisión: 0.06
 ingresos: \$600.00

salario base + empleado por comisión: Bob Lewis
 número de seguro social: 444-44-4444
 ventas brutas: \$5,000.00
 tarifa de comisión: 0.04; salario base: \$300.00
 ingresos: \$500.00

Empleados procesados en forma polimórfica:

empleado asalariado: John Smith
 número de seguro social: 111-11-1111
 salario semanal: \$800.00
 ingresos \$800.00

empleado por horas: Karen Price
 número de seguro social: 222-22-2222
 sueldo por horas: \$16.75; horas trabajadas: 40.00
 ingresos \$670.00

Figura 11.9 | Aplicación de prueba de la jerarquía de Empleado. (Parte 2 de 3).

```

empleado por comisión: Sue Jones
número de seguro social: 333-33-3333
ventas brutas: $10,000.00
tarifa de comisión: 0.06
ingresos $600.00

salario base + empleado por comisión: Bob Lewis
número de seguro social: 444-44-4444
ventas brutas: $5,000.00
tarifa de comisión: 0.04; salario base: $300.00
nuevo salario base con incremento del 10%: $330.00
ingresos $530.00

Empleado 0 es un EmpleadoAsalariado
Empleado 1 es un EmpleadoPorHoras
Empleado 2 es un EmpleadoPorComision
Empleado 3 es un EmpleadoBaseMasComision

```

Figura 11.9 | Aplicación de prueba de la jerarquía de Empleado. (Parte 3 de 3).

La línea 35 declara a `empleados` y le asigna un arreglo de cuatro variables `Empleado`. Las líneas 38-41 asignan un objeto `EmpleadoAsalariado`, un objeto `EmpleadoPorHoras`, un objeto `EmpleadoPorComision` y un objeto `EmpleadoBaseMasComision` a `empleados[0]`, `empleados[1]`, `empleados[2]` y `empleados[3]`, respectivamente. Cada asignación es permitida, ya que un `EmpleadoAsalariado` es un `Empleado`, un `EmpleadoPorHoras` es un `Empleado`, un `EmpleadoPorComision` es un `Empleado` y un `EmpleadoBaseMasComision` es un `Empleado`. Por lo tanto, podemos asignar las referencias de los objetos `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision` a variables de la clase base `Empleado`, aun cuando ésta es una clase abstracta.

Las líneas 46-66 iteran a través del arreglo `empleados` e invocan los métodos `ToString` e `Ingresos` con la variable `empleadoActual` de `Empleado`, a la cual se le asigna la referencia a un `Empleado` distinto en el arreglo, durante cada iteración. Los resultados ilustran que en definitivo se invocan los métodos apropiados para cada clase. Todas las llamadas a los métodos `ToString` e `Ingresos` se resuelven en tiempo de ejecución, con base en el tipo del objeto al que `empleadoActual` hace referencia. Este proceso se conoce como *vinculación dinámica* o *vinculación postergada*. Por ejemplo, la línea 48 invoca en forma implícita al método `ToString` del objeto al que `empleadoActual` hace referencia. Como resultado de la vinculación dinámica, el CLR decide qué método `ToString` de cuál clase va a llamar en tiempo de ejecución, en vez de hacerlo en tiempo de compilación. Observe que sólo los métodos de la clase `Empleado` pueden llamarse a través de una variable `Empleado`; y desde luego que `Empleado` incluye los métodos de la clase `object`, como `ToString`. (En la sección 10.7 vimos el conjunto de métodos que todas las clases herdan de la clase `object`). Una referencia a la clase base puede utilizarse para invocar sólo a métodos de la clase base.

Realizamos un procesamiento especial en los objetos `EmpleadoBasePorComision`; a medida que los encontramos, incrementamos su salario base en 10%. Cuando procesamos objetos en forma polimórfica, por lo general no necesitamos preocuparnos por los “detalles específicos”, pero para ajustar el salario base, tenemos que determinar el tipo específico de cada objeto `Empleado` en tiempo de ejecución. La línea 51 utiliza el operador `is` para determinar si el tipo de cierto objeto `Empleado` es `EmpleadoBaseMasComision`. La condición en la línea 51 es verdadera si el objeto al que hace referencia `empleadoActual` es un `EmpleadoBaseMasComision`. Esto también sería verdadero para cualquier objeto de una clase derivada de `EmpleadoBaseMasComision` (si hubiera alguna), debido a la relación *es un* que tiene una clase derivada con su clase base. Las líneas 55-65 realizan una conversión descendente en `empleadoActual`, del tipo `Empleado` al tipo `EmpleadoBaseMasComision`; esta conversión se permite sólo si el objeto tiene una relación *es un* con `EmpleadoBaseMasComision`. La condición en la línea 51 asegura que éste sea el caso. Esta conversión se requiere si vamos a utilizar la propiedad `SalarioBase` de la clase derivada `EmpleadoBaseMasComision` en el objeto `Empleado` actual; si tratamos de invocar a un método que pertenezca sólo a la clase derivada en una referencia a la clase base, se produce un error de compilación.



Error común de programación 11.3

Asignar una variable de la clase base a una variable de la clase derivada (sin una conversión descendente explícita) es un error de compilación.



Observación de ingeniería de software 11.4

Si en tiempo de ejecución se asigna la referencia a un objeto de la clase derivada a una variable de una de sus clases base directas o indirectas, es aceptable convertir la referencia almacenada en esa variable de la clase base, de vuelta a una referencia del tipo de la clase derivada. Antes de realizar dicha conversión, use el operador `is` para asegurar que el objeto sea de un tipo de clase derivada apropiado.



Error común de programación 11.4

Al realizar una conversión descendente sobre un objeto, se produce una excepción `InvalidCastException` (del espacio de nombres `System`) si, en tiempo de ejecución, el objeto no tiene una relación “es un” con el tipo especificado en el operador de conversión. Un objeto puede convertirse sólo a su propio tipo o al tipo de una de sus clases base.

Si la expresión `is` en la línea 51 es `true`, la instrucción `if` (líneas 51-62) realiza el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. Usando la variable `empleado` de `EmpleadoBaseMasComision`, la línea 58 utiliza la propiedad `SalarioBase`, que sólo pertenece a la clase derivada, para extraer y actualizar el salario base del empleado con el aumento del 10 por ciento.

Las líneas 64-65 invocan al método `Ingresos` en `empleadoActual`, el cual llama al método `Ingresos` del objeto de la clase derivada apropiada en forma polimórfica. Observe que al obtener en forma polimórfica los ingresos del `EmpleadoAsalariado`, el `EmpleadoPorHoras` y el `EmpleadoPorComision` en las líneas 64-65, se produce el mismo resultado que obtener los ingresos de estos empleados en forma individual, en las líneas 24-32. No obstante, el monto de los ingresos obtenidos para el `EmpleadoBaseMasComision` en las líneas 64-65 es más alto que el que se obtiene en las líneas 30-32, debido al aumento del 10% en su salario base.

Las líneas 69-71 imprimen en pantalla el tipo de cada empleado, como un objeto `string`. Todos los objetos en C# conocen su propia clase y pueden acceder a esta información a través del método `GetType`, que todas las clases heredan de la clase `object`. El método `GetType` devuelve un objeto de la clase `Type` (del espacio de nombres `System`), el cual contiene información acerca del tipo del objeto, incluyendo el nombre de su clase, los nombres de sus métodos `public` y el nombre de su clase base. La línea 71 invoca al método `GetType` en el objeto para obtener su clase en tiempo de ejecución (es decir, un objeto `Type` que representa el tipo del objeto). Después el método `ToString` se invoca en forma implícita en el objeto devuelto por `GetType`. El método `ToString` de la clase `Type` devuelve el nombre de la clase.

En el ejemplo anterior, evitamos varios errores de compilación mediante la conversión descendente de la variable `Empleado` a una variable `EmpleadoBaseMasComision` en las líneas 55-56. Si eliminamos el operador de conversión (`EmpleadoBaseMasComision`) de la línea 56 y tratamos de asignar la variable `empleadoActual` de `Empleado` directamente a la variable `empleado` de `EmpleadoBaseMasComision`, recibiremos un error de compilación del tipo “*No se puede convertir implícitamente al tipo*”. Este error indica que el intento de asignar la referencia del objeto `EmpleadoPorComision` de la clase base a la variable `EmpleadoBaseMasComision` de la clase derivada no se permite sin un operador de conversión apropiado. El compilador evita esta asignación debido a que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`; de nuevo, la relación *es un* se aplica sólo entre la clase derivada y sus clases base, no viceversa.

De manera similar, si las líneas 58 y 61 utilizan la variable `empleadoActual` de la clase base, en vez de la variable `empleado` de la clase derivada, para usar la propiedad `SalarioBase` que pertenece sólo a la clase derivada, recibiríamos un error de compilación del tipo “*'Empleado' no contiene una definición para 'SalarioBase'*” en cada una de estas líneas. No se permite tratar de invocar métodos que pertenezcan sólo a la clase derivada en una referencia a la clase base. Mientras que las líneas 58 y 61 se ejecutan sólo si la instrucción `is` en la línea 51 devuelve `true` para indicar que a `empleadoActual` se le asignó una referencia a un objeto `EmpleadoBaseMasComision`, no podemos tratar de usar la propiedad `SalarioBase` de la clase derivada `EmpleadoBaseMasComision` con la referencia `empleadoActual` de la clase base `Empleado`. El compilador generaría errores en las líneas 58 y 61, ya que `SalarioBase` no es miembro de la clase base y no puede utilizarse con una variable de la clase base. Aunque el método que se vaya a llamar en realidad depende del tipo del objeto en tiempo de ejecución, puede

utilizarse una variable para invocar sólo a los métodos que sean miembros del tipo de esa variable, lo cual verifica el compilador. Si utilizamos una variable `Empleado` de la clase base, sólo podemos invocar a los métodos y propiedades que se encuentran en la clase `Empleado`: los métodos `Ingresos` y `ToString`, y las propiedades `PrimerNombre`, `ApellidoPaterno` y `NumeroSeguroSocial`.

11.5.7 Resumen de las asignaciones permitidas entre variables de la clase base y de la clase derivada

Ahora que hemos visto una aplicación completa que procesa diversos objetos de clases derivadas en forma polimórfica, sintetizaremos lo que puede y no puede hacer con los objetos y variables de las clases base y las clases derivadas. Aunque un objeto de una clase derivada también es un objeto de su clase base, los dos objetos son, sin embargo, distintos. Como vimos antes, los objetos de una clase derivada pueden tratarse como si fueran objetos de la clase base. No obstante, la clase derivada puede tener miembros adicionales que sólo pertenezcan a esa clase. Por esta razón, no se permite asignar una referencia de clase base a una variable de clase derivada sin una conversión explícita; dicha asignación dejaría los miembros de la clase derivada indefinidos para un objeto de la clase base.

Hemos visto cuatro maneras de asignar referencias de clase base y de clase derivada a las variables de los tipos de clase base y clase derivada:

1. Asignar una referencia de clase base a una variable de clase base es un proceso simple y directo.
2. Asignar una referencia de clase derivada a una variable de clase derivada es un proceso simple y directo.
3. Asignar una referencia de clase derivada a una variable de clase base es seguro, ya que el objeto de la clase derivada es un objeto de su clase base. No obstante, esta referencia puede usarse para referirse sólo a los miembros de la clase base. Si este código hace referencia a los miembros que pertenezcan sólo a la clase derivada, a través de la variable de la clase base, el compilador reporta errores.
4. Tratar de asignar una referencia de clase base a una variable de clase derivada es un error de compilación. Para evitar este error, la referencia de clase base debe convertirse en forma explícita a un tipo de clase derivada. En tiempo de ejecución, si el objeto al que se refiere la referencia no es un objeto de la clase derivada, se producirá una excepción. (Para más información sobre el manejo de excepciones, vea el capítulo 12, Manejo de excepciones). El operador `is` puede utilizarse para asegurar que dicha conversión se realice sólo si el objeto es de la clase derivada.

11.6 Métodos y clases sealed

En la sección 10.4 vimos que sólo pueden redefinirse en las clases derivadas los métodos que se declaran como `virtual`, `override` o `abstract`. Un método que se declara como `sealed` en una clase base no puede redefinirse en una clase derivada. Los métodos que se declaran como `private` son `sealed` de manera implícita, ya que es imposible redefinirlos en una clase derivada (aunque la clase derivada puede declarar un nuevo método con la misma firma que el método `private` en la clase base). Los métodos que se declaran como `static` también son `sealed` de manera implícita, ya que los métodos `static` tampoco pueden redefinirse. Un método de clase derivada que se declare tanto `override` como `sealed` puede redefinir a un método de la clase base, pero no puede redefinirse en clases derivadas que se encuentren a niveles más bajos de la jerarquía de herencia.

La declaración de un método `sealed` no puede cambiar nunca, por lo que todas las clases derivadas utilizan la misma implementación del método, y las llamadas a los métodos `sealed` se resuelven en tiempo de compilación; a esto se le conoce como *vinculación estática*. Como el compilador sabe que los métodos `sealed` no pueden ser redefinidos, a menudo puede optimizar el código eliminando llamadas a los métodos `sealed` y sustituyéndolas con el código expandido de sus declaraciones en cada ubicación de llamada al método; a esta técnica se le conoce como *poner el código en línea*.



Tip de rendimiento 11.1

El compilador puede decidir si va a poner en línea la llamada a un método `sealed`, y lo hará para los métodos `sealed` pequeños y simples. Poner en línea el código de una llamada a un método no viola el encapsulamiento ni el ocultamiento de información y aumenta el rendimiento, ya que elimina la sobrecarga en la que se incurre al hacer una llamada a un método.

Una clase que se declara como `sealed` no puede ser una clase base (es decir, una clase no puede extender a una clase `sealed`). Todos los métodos en una clase `sealed` son `sealed` de manera implícita. La clase `string` es una clase `sealed`. Esta clase no puede extenderse, por lo que las aplicaciones que utilizan objetos `string` pueden depender de la funcionalidad de los objetos `string` que se especifica en la FCL.



Error común de programación 11.5

Tratar de declarar una clase derivada de una clase sealed es un error de compilación.



Observación de ingeniería de software 11.5

En la FCL, la vasta mayoría de clases no se declaran como sealed. Esto permite el uso de la herencia y el polimorfismo: las herramientas fundamentales de la programación orientada a objetos.

11.7 Caso de estudio: creación y uso de interfaces

En nuestro siguiente ejemplo (figuras 11.11 a 11.15) utilizaremos otra vez el sistema de nómina de la sección 11.5. Suponga que la compañía involucrada desea realizar varias operaciones de contabilidad en una sola aplicación de cuentas por pagar; además de calcular los ingresos de nómina que deben pagarse a cada empleado, la compañía debe también calcular el pago vencido en cada una de varias facturas (por los bienes comprados). Aunque se aplican a cosas no relacionadas (es decir, empleados y facturas), ambas operaciones tienen que ver con el cálculo de algún tipo de monto a pagar. Para un empleado, el pago se refiere a los ingresos del empleado. Para una factura, el pago se refiere al costo total de los bienes listados en la factura. ¿Podemos calcular esas cosas distintas, tales como los pagos vencidos para los empleados y las facturas, en forma polimórfica en una sola aplicación? ¿Ofrece C# una capacidad que requiera que las clases no relacionadas implementen un conjunto de métodos comunes (por ejemplo, un método que calcule un monto a pagar)? Las interfaces de C# ofrecen exactamente esta capacidad.

Las interfaces definen y estandarizan las formas en que pueden interactuar las personas y los sistemas entre sí. Por ejemplo, los controles en un radio sirven como una interfaz entre los usuarios del radio y sus componentes internos. Los controles permiten a los usuarios realizar un conjunto limitado de operaciones (por ejemplo, cambiar la estación, ajustar el volumen, seleccionar AM o FM), y distintos radios pueden implementar los controles de distintas formas (por ejemplo, el uso de botones, perillas, comandos de voz). La interfaz especifica *qué* operaciones debe permitir el radio que realicen los usuarios, pero no especifica *cómo* deben realizarse las operaciones. De manera similar, la interfaz entre un conductor y un auto con transmisión manual incluye el volante, la palanca de cambios, el pedal del embrague, el pedal del acelerador y el pedal del freno. Esta misma interfaz se encuentra en casi todos los autos de transmisión manual, lo que permite que alguien que sabe cómo manejar cierto auto de transmisión manual sepa cómo manejar casi cualquier auto de transmisión manual. Los componentes de cada auto individual pueden tener una apariencia ligeramente distinta, pero el propósito general es el mismo; permitir que las personas conduzcan el auto.

Los objetos de software también se comunican a través de interfaces. Una interfaz de C# describe un conjunto de métodos que pueden llamarse sobre un objeto, para indicar al objeto que realice cierta tarea, o que devuelva cierta pieza de información, por ejemplo. El siguiente ejemplo introduce una interfaz llamada `IPorPagar`, la cual describe la funcionalidad de cualquier objeto que deba ser capaz de recibir un pago y, por lo tanto, debe ofrecer un método para determinar el monto de pago vencido apropiado. La *declaración de una interfaz* empieza con la palabra clave `interface` y sólo puede contener métodos, propiedades, indexadores y eventos (hablaremos sobre los eventos en el capítulo 13, Conceptos de interfaz gráfica de usuario: parte I) abstractos. Todos los miembros de la interfaz se declaran en forma implícita como `public` y `abstract`. Además, cada interfaz puede extender a otra interfaz o más interfaces, para crear una interfaz más elaborada que otras clases puedan implementar.



Error común de programación 11.6

Es un error de compilación declarar un miembro de la interfaz public o abstract en forma explícita, debido a que es redundante en las declaraciones de miembros de interfaces. También es un error de compilación especificar cualquier detalle de implementación, tal como las declaraciones de métodos concretos, en una interfaz.

Para utilizar una interfaz, una clase debe especificar que *implementa* a esa interfaz mediante un listado de esa interfaz después del signo de dos puntos (:) en la declaración de la clase. Observe que ésta es la misma sintaxis que se utiliza para indicar la herencia de una clase base. Una clase concreta que implementa a esta interfaz debe declarar cada miembro de la interfaz con la firma especificada en la declaración de la interfaz. Una clase que implemente a una interfaz, pero que no implementa a todos los miembros de la misma, es una clase abstracta; debe declararse como `abstract` y debe contener una declaración `abstract` para cada miembro de la interfaz que no implemente. Implementar una interfaz es como firmar un contrato con el compilador que diga, “Proporcionaré una implementación para todos los miembros especificados por la interfaz, o los declararé como `abstract`”.



Error común de programación 11.7

Si no declaramos ningún miembro de una interfaz, en una clase que implemente a esa interfaz, se produce un error de compilación.

Por lo general, una interfaz se utiliza cuando clases dispares (es decir, no relacionadas) necesitan compartir métodos comunes. Esto permite que los objetos de clases no relacionadas se procesen en forma polimórfica; los objetos de clases que implementan la misma interfaz pueden responder a las mismas llamadas a métodos. Los programadores pueden crear una interfaz que describa la funcionalidad deseada y después implementar esta interfaz en cualquier clase que requiera esa funcionalidad. Por ejemplo, en la aplicación de cuentas por pagar que desarrollaremos en esta sección, implementaremos la interfaz `IPorPagar` en cualquier clase que deba tener la capacidad de calcular el monto de un pago (por ejemplo, `Empleado`, `Factura`).

A menudo, una interfaz se utiliza en vez de una clase `abstract` cuando no hay una implementación predeterminada que heredar; esto es, no hay campos ni implementaciones de métodos predeterminadas. Al igual que las clases `public abstract`, las interfaces son comúnmente de tipo `public`, por lo que se declaran en archivos por sí solas con el mismo nombre que la interfaz, y la extensión de archivo `.cs`.

11.7.1 Desarrollo de una jerarquía `IPorPagar`

Para crear una aplicación que pueda determinar los pagos para los empleados y facturas por igual, primero vamos a crear una interfaz llamada `IPorPagar`. Esta interfaz contiene el método `ObtenerMontoPago`, el cual devuelve un monto decimal que debe pagarse para un objeto de cualquier clase que implemente a la interfaz. El método `ObtenerMontoPago` es una versión de propósito general del método `Ingresos` de la jerarquía de `Empleado`; el método `Ingresos` calcula un monto de pago específicamente para un `Empleado`, mientras que `ObtenerMontoPago` puede aplicarse a un amplio rango de objetos no relacionados. Después de declarar la interfaz `IPorPagar` presentaremos la clase `Factura`, la cual implementa a la interfaz `IPorPagar`. Luego modificaremos la clase `Empleado` de tal forma que también implemente a la interfaz `IPorPagar`. Por último, actualizaremos la clase `EmpleadoAsalariado` derivada de `Empleado` para “ajustarla” en la jerarquía de `IPorPagar` (es decir, cambiaremos el nombre del método `Ingresos` de `EmpleadoAsalariado` por el de `ObtenerMontoPago`).



Buena práctica de programación 11.1

Por convención, el nombre de una interfaz empieza con "I". Esto ayuda a diferenciar las interfaces de las clases, con lo cual se mejora la legibilidad del código.



Buena práctica de programación 11.2

Al declarar un método en una interfaz, seleccione un nombre para el método que describa su propósito en forma general, ya que el método podría implementarse por un amplio rango de clases no relacionadas.

Las clases `Factura` y `Empleado` representan cosas para las cuales la compañía debe calcular un monto a pagar. Ambas clases implementan a `IPorPagar`, por lo que una aplicación puede invocar al método `ObtenerMontoPago` en objetos `Factura` y `Empleado` por igual. Esto permite el procesamiento polimórfico de objetos `Factura` y `Empleado` requerido para la aplicación de cuentas por pagar de nuestra compañía.

El diagrama de clases de UML en la figura 11.10 muestra la jerarquía de interfaz y clases utilizada en nuestra aplicación de cuentas por pagar. La jerarquía comienza con la interfaz `IPorPagar`. UML diferencia a una interfaz de una clase colocando la palabra “interface” entre los signos «» y », por encima del nombre de la interfaz. UML

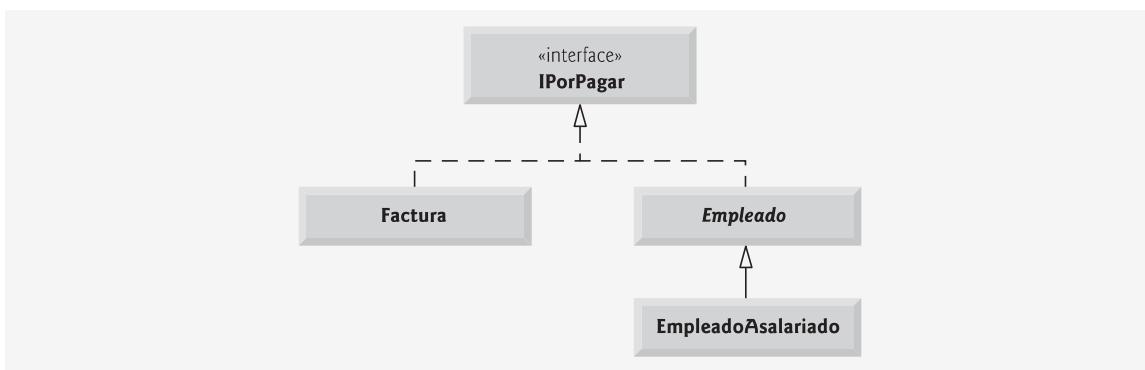


Figura 11.10 | Diagrama de clases de UML de la jerarquía de clases y la interfaz IPorPagar.

```

1 // Fig. 11.11: IPorPagar.cs
2 // Declaración de la interfaz IPorPagar.
3 public interface IPorPagar
4 {
5     decimal ObtenerMontoPago(); // calcula el pago; no hay implementación
6 } // fin de la interfaz IPorPagar
  
```

Figura 11.11 | Declaración de la interfaz IPorPagar.

expresa la relación entre una clase y una interfaz a través de una *realización*. Se dice que una clase “realiza”, o implementa, a una interfaz. Un diagrama de clases modela una realización como una flecha punteada que parte de la clase que va a realizar la implementación, hasta la interfaz. El diagrama en la figura 11.10 indica que cada una de las clases Factura y Empleado realizan (es decir, implementan) la interfaz IPorPagar. Observe que, al igual que en el diagrama de clases de la figura 11.2, la clase Empleado aparece en cursivas, lo cual indica que es una clase abstracta. La clase concreta EmpleadoAsalariado extiende a Empleado y hereda la relación de realización de la clase base con la interfaz IPorPagar.

11.7.2 Declaración de la interfaz IPorPagar

La declaración de la interfaz IPorPagar empieza en la figura 11.11, línea 3. La interfaz IPorPagar contiene el método `public abstract ObtenerMontoPago()` (línea 5). Recuerde que este método no puede declararse en forma explícita como `public` o `abstract`. La interfaz IPorPagar sólo tiene un método, pero las interfaces pueden tener cualquier número de miembros. Además, el método `ObtenerMontoPago` no tiene parámetros, pero los métodos de las interfaces pueden tener parámetros.

11.7.3 Creación de la clase Factura

Ahora vamos a crear la clase Factura (figura 11.12) para representar una factura simple que contiene información de facturación para cierto tipo de pieza. La clase declara las variables de instancia `private numeroPieza`, `descripcionPieza`, `cantidad` y `precioPorArticulo` (líneas 5-8), las cuales indican el número de pieza, su descripción, la cantidad de piezas ordenadas y el precio por artículo. La clase Factura también contiene un constructor (líneas 11-18), propiedades (líneas 21-70) que manipulan las variables de instancia de la clase y un método `ToString()` (líneas 73-79) que devuelve una representación `string` de un objeto Factura. Observe que los descriptores de acceso `set` de las propiedades `Cantidad` (líneas 53-56) y `PrecioPorArticulo` (líneas 66-69) aseguran que a `cantidad` y a `precioPorArticulo` se les asignen sólo valores no negativos.

La línea 3 de la figura 11.12 indica que la clase Factura implementa a la interfaz IPorPagar. Al igual que todas las clases, la clase Factura también extiende a `object` de manera implícita. C# no permite que las clases derivadas hereden de más de una clase base, pero sí permite que una clase herede de una clase base e implemente cualquier número de interfaces. Todos los objetos de una clase que implementan varias interfaces tienen la

relación *es un* con cada tipo de interfaz implementada. Para implementar más de una interfaz, utilice una lista separada por comas de nombres de interfaz después del signo de dos puntos (:) en la declaración de la clase, como se muestra a continuación:

```
public class NombreClase : NombreClaseBase, PrimeraInterfaz, SegundaInterfaz, ...
```

Cuando una clase hereda de una clase base e implementa a una o más interfaces, la declaración de la clase debe listar el nombre de la clase base antes de cualquier nombre de interfaz.

```

1 // Fig. 11.12: Factura.cs
2 // La clase Factura implementa a IPorPagar.
3 public class Factura : IPorPagar
4 {
5     private string numeroPieza;
6     private string descripcionPieza;
7     private int cantidad;
8     private decimal precioPorArticulo;
9
10    // constructor con cuatro parámetros
11    public Factura( string pieza, string descripcion, int cuenta,
12                    decimal precio )
13    {
14        NumeroPieza = pieza;
15        DescripcionPieza = descripcion;
16        Cantidad = cuenta; // valida la cantidad a través de una propiedad
17        PrecioPorArticulo = precio; // valida el precio por artículo a través de una
18        // propiedad
19    } // fin del constructor de Factura con cuatro parámetros
20
21    // propiedad que obtiene y establece el número de pieza en la factura
22    public string NumeroPieza
23    {
24        get
25        {
26            return numeroPieza;
27        } // fin de get
28        set
29        {
30            numeroPieza = value; // debería validarse
31        } // fin de set
32    } // fin de propiedad NumeroPieza
33
34    // propiedad que obtiene y establece la descripción de la pieza en la factura
35    public string DescripcionPieza
36    {
37        get
38        {
39            return descripcionPieza;
40        } // fin de get
41        set
42        {
43            descripcionPieza = value; // debería validarse
44        } // fin de set
45    } // fin de la propiedad DescripcionPieza
46
47    // propiedad que obtiene y establece la cantidad en la factura
48    public int Cantidad

```

Figura 11.12 | La clase Factura implementa a IPorPagar. (Parte 1 de 2).

```

48     {
49         get
50         {
51             return cantidad;
52         } // fin de get
53         set
54         {
55             cantidad = ( value < 0 ) ? 0 : value; // valida la cantidad
56         } // fin de set
57     } // fin de la propiedad Cantidad
58
59     // propiedad que obtiene y establece el precio por artículo
60     public decimal PrecioPorArticulo
61     {
62         get
63         {
64             return precioPorArticulo;
65         } // fin de get
66         set
67         {
68             precioPorArticulo = ( value < 0 ) ? 0 : value; // valida el precio
69         } // fin de set
70     } // fin de la propiedad PrecioPorArticulo
71
72     // devuelve representación string de objeto Factura
73     public override string ToString()
74     {
75         return string.Format(
76             "{0}: \n{1}: {2} ({3}) \n{4}: {5} \n{6}: {7:C}",
77             "factura", "número de pieza", NumeroPieza, DescripcionPieza,
78             "cantidad", Cantidad, "precio por artículo", PrecioPorArticulo );
79     } // fin del método ToString
80
81     // método requerido para llevar a cabo el contrato con la interfaz IPorPagar
82     public decimal ObtenerMontoPago()
83     {
84         return Cantidad * PrecioPorArticulo; // calcula el costo total
85     } // fin del método ObtenerMontoPago
86 } // fin de la clase Factura

```

Figura 11.12 | La clase Factura implementa a IPorPagar. (Parte 2 de 2).

La clase Factura implementa el único método de la interfaz IPorPagar; el método ObtenerMontoPago se declara en las líneas 82-85. Este método calcula el monto requerido para pagar la factura. El método multiplica los valores de cantidad y precioPorArtículo (que se obtienen a través de las propiedades apropiadas) y devuelve el resultado (línea 84). Este método cumple con el requerimiento de implementación para el método en la interfaz IPorPagar; hemos cumplido el contrato de interfaz con el compilador.

11.7.4 Modificación de la clase Empleado para implementar la interfaz IPorPagar

Ahora vamos a modificar la clase Empleado para que implemente la interfaz IPorPagar. La figura 11.13 contiene la clase Empleado modificada. Esta declaración de la clase es idéntica a la de la figura 11.4, con dos excepciones. En primer lugar, la línea 3 de la figura 11.13 indica que la clase Empleado ahora implementa a la interfaz IPorPagar. En segundo lugar, como Empleado ahora implementa a la interfaz IPorPagar, debemos cambiar el nombre de Ingresos por el de ObtenerMontoPago en toda la jerarquía de Empleado. Sin embargo, al igual que con el método Ingresos en la versión de la clase Empleado de la figura 11.4, no tiene sentido implementar el método ObtenerMontoPago en la clase Empleado, ya que no podemos calcular el pago de los ingresos para un Empleado general; primero debemos conocer el tipo específico de Empleado. En la figura 11.4 declaramos el método

Ingresos como `abstract` por esta razón y, como resultado, la clase `Empleado` tuvo que declararse como `abstract`. Esto obliga a cada clase derivada de `Empleado` a redefinir `Ingresos` con una implementación concreta.

En la figura 11.13, manejamos esta situación de la misma forma. Recuerde que cuando una clase implementa a una interfaz, la clase hace un contrato con el compilador, en el que se establece que la clase va a implementar cada uno de los métodos en la interfaz, o los va a declarar como `abstract`. Si se elige la última opción, también debemos declarar la clase como `abstract`. Como vimos en la sección 11.4, cualquier clase derivada concreta de la clase abstracta debe implementar a los métodos `abstract` de la clase base. Si la clase derivada no lo hace, también debe declararse como `abstract`. Como lo indican los comentarios en las líneas 51-52, la clase `Empleado` de la figura 11.13 no implementa al método `ObtenerMontoPago`, por lo que la clase se declara como `abstract`.

```

1 // Fig. 11.13: Empleado.cs
2 // La clase base abstracta Empleado.
3 public abstract class Empleado : IPorPagar
4 {
5     private string primerNombre;
6     private string apellidoPaterno;
7     private string numeroSeguroSocial;
8
9     // constructor con tres parámetros
10    public Empleado( string nombre, string apellido, string nss )
11    {
12        primerNombre = nombre;
13        apellidoPaterno = apellido;
14        numeroSeguroSocial = nss;
15    } // fin del constructor de Empleado con tres parámetros
16
17    // propiedad de sólo lectura que obtiene el primer nombre del empleado
18    public string PrimerNombre
19    {
20        get
21        {
22            return primerNombre;
23        } // fin de get
24    } // fin de la propiedad PrimerNombre
25
26    // propiedad de sólo lectura que obtiene el apellido paterno del empleado
27    public string ApellidoPaterno
28    {
29        get
30        {
31            return apellidoPaterno;
32        } // fin de get
33    } // fin de la propiedad ApellidoPaterno
34
35    // propiedad de sólo lectura que obtiene el número de seguro social del empleado
36    public string NumeroSeguroSocial
37    {
38        get
39        {
40            return numeroSeguroSocial;
41        } // fin de get
42    } // fin de la propiedad NumeroSeguroSocial
43
44    // devuelve representación string del objeto Empleado
45    public override string ToString()

```

Figura 11.13 | La clase base abstracta `Empleado`. (Parte 1 de 2).

```

46     {
47         return string.Format( "{0} {1}\nNúmero de seguro social: {2}",
48             PrimerNombre, ApellidoPaterno, NumeroSeguroSocial );
49     } // fin del método ToString
50
51     // Nota: No implementamos aquí el método ObtenerMontoPago de IPorPagar, por lo
52     // que esta clase debe declararse como abstracta para evitar un error de compilación.
53     public abstract decimal ObtenerMontoPago();
54 } // fin de la clase abstracta Empleado

```

Figura 11.13 | La clase base abstracta Empleado. (Parte 2 de 2).

11.7.5 Modificación de la clase EmpleadoAsalariado para usarla en la jerarquía IPorPagar

La figura 11.14 contiene una versión modificada de la clase EmpleadoAsalariado que extiende a Empleado e implementa el método ObtenerMontoPago. Esta versión de EmpleadoAsalariado es idéntica a la de la figura 11.5, con la excepción de que esta versión implementa al método ObtenerMontoPago (líneas 29-32) en vez del método Ingresos. Los dos métodos contienen la misma funcionalidad, pero tienen distintos nombres. Recuerde que la versión de IPorPagar del método tiene un nombre más general para que pueda aplicarse a clases que sean posiblemente dispares. El resto de las clases derivadas de Empleado (EmpleadoPorHoras, EmpleadoPorComision y EmpleadoBaseMasComision) también deben modificarse para que contengan el método ObtenerMontoPago en vez de Ingresos, y así reflejar el hecho de que ahora Empleado implementa a IPorPagar. Dejaremos estas modificaciones como un ejercicio y sólo utilizaremos a EmpleadoAsalariado en nuestra aplicación de prueba en esta sección.

```

1  // Fig. 11.14: EmpleadoAsalariado.cs
2  // La clase EmpleadoAsalariado que extiende a Empleado.
3  public class EmpleadoAsalariado : Empleado
4  {
5      private decimal salarioSemanal;
6
7      // constructor con cuatro parámetros
8      public EmpleadoAsalariado( string nombre, string apellido, string nss,
9          decimal salario ) : base( nombre, apellido, nss )
10     {
11         SalarioSemanal = salario; // valida el salario a través de una propiedad
12     } // fin del constructor de EmpleadoAsalariado con cuatro parámetros
13
14     // propiedad que obtiene y establece el salario del empleado asalariado
15     public decimal SalarioSemanal
16     {
17         get
18         {
19             return salarioSemanal;
20         } // fin de get
21         set
22         {
23             salarioSemanal = value < 0 ? 0 : value; // validación
24         } // fin de set
25     } // fin de la propiedad SalarioSemanal
26
27     // calcula los ingresos; implementa el método de la interfaz
28     // IPorPagar que es abstracto en la clase base Empleado

```

Figura 11.14 | La clase EmpleadoAsalariado que extiende a Empleado. (Parte 1 de 2).

```

29  public override decimal ObtenerMontoPago()
30  {
31      return SalarioSemanal;
32  } // fin del método ObtenerMontoPago
33
34  // devuelve representación string del objeto EmpleadoAsalariado
35  public override string ToString()
36  {
37      return string.Format("empleado asalariado: {0}\n{1}: {2:C}",
38          base.ToString(), "salario semanal", SalarioSemanal );
39  } // fin del método ToString
40 } // fin de la clase EmpleadoAsalariado

```

Figura 11.14 | La clase EmpleadoAsalariado que extiende a Empleado. (Parte 2 de 2).

Cuando una clase implementa a una interfaz, se aplica la misma relación *es un* que proporciona la herencia. Por ejemplo, la clase Empleado implementa a IPorPagar, por lo que podemos decir que un Empleado es un IPorPagar, al igual que cualquier clase que extienda a Empleado. Por ejemplo, los objetos EmpleadoAsalariado son objetos IPorPagar. Al igual que con las relaciones de herencia, un objeto de una clase que implemente a una interfaz puede considerarse como un objeto del tipo de la interfaz. Los objetos de cualquier clase derivada de la clase que implementa a la interfaz también pueden considerarse como objetos del tipo de la interfaz. Por ende, así como podemos asignar la referencia de un objeto EmpleadoAsalariado a una variable de la clase base Empleado, también podemos asignar la referencia de un objeto EmpleadoAsalariado a una interfaz IPorPagar. Factura implementa a IPorPagar, por lo que un objeto Factura también *es un* objeto IPorPagar, y podemos asignar la referencia de un objeto Factura a una variable IPorPagar.



Observación de ingeniería de software 11.6

La herencia y las interfaces son similares en cuanto a su implementación de la relación es un. Un objeto de una clase que implementa a una interfaz puede considerarse como un objeto del tipo de esa interfaz. Un objeto de cualquier clase derivada de una clase que implementa a una interfaz también puede considerarse como un objeto del tipo de la interfaz.



Observación de ingeniería de software 11.7

La relación es un que existe entre las clases base y las clases derivadas, y entre las interfaces y las clases que las implementan, se mantiene cuando se pasa un objeto a un método. Cuando el parámetro de un método recibe una variable de una clase base o de un tipo de interfaz, el método procesa en forma polimórfica al objeto que recibe como argumento.

11.7.6 Uso de la interfaz IPorPagar para procesar objetos Factura y Empleado mediante el polimorfismo

PruebaInterfazPorPagar (figura 11.15) ilustra que la interfaz IPorPagar puede usarse para procesar un conjunto de objetos Factura y Empleado en forma polimórfica en una sola aplicación. La línea 10 declara a objetosPorPagar y le asigna un arreglo de cuatro variables IPorPagar. Las líneas 13-14 asignan las referencias de objetos EmpleadoAsalariado a los dos elementos restantes de objetosPorPagar. Las líneas 15-18 asignan las referencias a los objetos EmpleadoSalario a los dos elementos restantes de objetosPorPagar. Estas asignaciones se permiten debido a que un objeto Factura *es un* objeto IPorPagar. Las líneas 24-29 utilizan una instrucción foreach para procesar cada objeto IPorPagar en objetosPorPagar de manera polimórfica, imprimiendo en pantalla el objeto como un string, junto con el pago vencido. Observe que la línea 27 invoca en forma implícita al método ToString de una referencia de la interfaz IPorPagar, aun cuando ToString no se declara en la interfaz IPorPagar; todas las referencias (incluyendo las de los tipos de interfaces) se refieren a objetos que extienden a object y, por lo tanto, tienen un método ToString. La línea 28 invoca al método ObtenerMontoPago de IPorPagar para obtener

el monto a pagar para cada objeto en `objetosPorPagar`, sin importar el tipo del objeto. Los resultados revelan que las llamadas a los métodos en las líneas 27-28 invocan a la implementación de la clase apropiada de los métodos `ToString` y `ObtenerMontoPago`. Por ejemplo, cuando `empleadoActual` se refiere a un objeto `Factura` durante la primera iteración del ciclo `foreach`, se ejecutan los métodos `ToString` y `ObtenerMontoPago` de la clase `Factura`.



Observación de ingeniería de software 11.8

Todos los métodos de la clase `object` pueden llamarse mediante el uso de una referencia de un tipo de interfaz; la referencia se refiere a un objeto, y todos los objetos heredan los métodos de la clase `object`.

```

1 // Fig. 11.15: PruebaInterfazPorPagar.cs
2 // Prueba la interfaz IPorPagar con clases dispares.
3 using System;
4
5 public class PruebaInterfazPorPagar
6 {
7     public static void Main( string[] args )
8     {
9         // crea arreglo IPorPagar de cuatro elementos
10        IPorPagar[] objetosPorPagar = new IPorPagar[ 4 ];
11
12        // llena el arreglo con objetos que implementan a IPorPagar
13        objetosPorPagar[ 0 ] = new Factura( "01234", "asiento", 2, 375.00M );
14        objetosPorPagar[ 1 ] = new Factura( "56789", "llanta", 4, 79.95M );
15        objetosPorPagar[ 2 ] = new EmpleadoAsalariado( "John", "Smith",
16            "111-11-1111", 800.00M );
17        objetosPorPagar[ 3 ] = new EmpleadoAsalariado( "Lisa", "Barnes",
18            "888-88-8888", 1200.00M );
19
20        Console.WriteLine(
21            "Facturas y Empleados procesados en forma polimórfica:\n" );
22
23        // procesa en forma genérica cada elemento en el arreglo objetosPorPagar
24        foreach ( IPorPagar porPagarActual in objetosPorPagar )
25        {
26            // imprime en pantalla el valor de porPagarActual y el monto a pagar apropiado
27            Console.WriteLine( "[0] \n{1}: {2:C}\n", porPagarActual,
28                "pago vencido", porPagarActual.ObtenerMontoPago() );
29        } // fin de foreach
30    } // fin de Main
31 } // fin de la clase PruebaInterfazPorPagar

```

Facturas y Empleados procesados en forma polimórfica:

```

factura:
número de pieza: 01234 (asiento)
cantidad: 2
precio por artículo: $375.00
pago vencido: $750.00

factura:
número de pieza: 56789 (llanta)
cantidad: 4
precio por artículo: $79.95
pago vencido: $319.80

```

Figura 11.15 | Prueba de la interfaz `IPorPagar` con clases dispares. (Parte 1 de 2).

```

empleado asalariado: John Smith
número de seguro social: 111-11-1111
salario semanal: $800.00
pago vencido: $800.00

empleado asalariado: Lisa Barnes
número de seguro social: 888-88-8888
salario semanal: $1,200.00
pago vencido: $1,200.00

```

Figura 11.15 | Prueba de la interfaz `IPorPagar` con clases dispares. (Parte 2 de 2).

11.7.7 Interfaces comunes de la Biblioteca de clases del .NET Framework

En esta sección veremos las generalidades acerca de varias interfaces comunes en la Biblioteca de clases del .NET Framework. Estas interfaces se implementan y se utilizan de la misma forma que la interfaz que creamos en secciones anteriores (la interfaz `IPorPagar` en la sección 11.7.2). Las interfaces de la FCL le permiten extender muchos aspectos importantes de C# con sus propias clases. La figura 11.16 muestra los aspectos generales de varias interfaces de uso común de la FCL.

11.8 Sobre carga de operadores

Las manipulaciones en los objetos de las clases se realizan mediante el envío de mensajes (en forma de llamadas a métodos) a los objetos. Esta notación de llamadas a métodos es incómoda para ciertos tipos de clases, en especial las clases matemáticas. Para estas clases, sería conveniente utilizar el robusto conjunto de operadores integrados de C# para especificar las manipulaciones de objetos. En esta sección le mostraremos cómo permitir que estos operadores trabajen con objetos de clases, a través de un proceso conocido como *sobre carga de operadores*.

Interface	Descripción
<code>IComparable</code>	Como vio en el capítulo 3, C# contiene varios operadores de comparación (por ejemplo, <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>!=</code>) que le permiten comparar valores de tipos simples. En la sección 11.8 verá que estos operadores se pueden definir para comparar dos objetos. La interfaz <code>IComparable</code> también puede utilizarse para permitir que los objetos de una clase que implementa a la interfaz se comparan unos con otros. La interfaz contiene un método, <code>CompareTo</code> , que compara el objeto que llama al método con el objeto que se pasa como argumento al método. Las clases deben implementar a <code>CompareTo</code> para devolver un valor que indica si el objeto en el cual se invoca es menor que (valor de retorno entero negativo), igual que (valor de retorno 0) o mayor que (valor de retorno entero positivo) el objeto que se pasa como argumento, usando cualquier criterio especificado por el programador. Por ejemplo, si la clase <code>Empleado</code> implementa a <code>IComparable</code> , su método <code>CompareTo</code> podría comparar objetos <code>Empleado</code> en base a sus montos de ingresos. La interfaz <code>IComparable</code> se utiliza comúnmente para ordenar objetos en una colección tal como un arreglo. En el capítulo 25, Genéricos y en el capítulo 26, Colecciones, utilizaremos la interfaz <code>IComparable</code> .
<code>IComponent</code>	Cualquier clase que represente un componente puede implementar a esta interfaz, incluyendo los controles (tales como botones o etiquetas) de la Interfaz Gráfica de Usuario (GUI). La interfaz <code>IComponent</code> define los comportamientos que deben implementar los componentes. En el capítulo 13, Conceptos de interfaz gráfica de usuario: parte I, y en el capítulo 14, Conceptos de interfaz gráfica de usuario: parte 2, hablaremos sobre <code>IComponent</code> y muchos controles de GUI que implementan a esta interfaz.

Figura 11.16 | Interfaces comunes de la Biblioteca de clases del .NET Framework. (Parte 1 de 2).

Interface	Descripción
IDisposable	Las clases que deben proporcionar un mecanismo explícito para liberar recursos pueden implementar a esta interfaz. Algunos recursos pueden ser utilizados sólo por un programa a la vez. Además, algunos recursos tales como los archivos en disco son recursos no administrados que, a diferencia de la memoria, no pueden ser liberados por el recolector de basura. Las clases que implementan a la interfaz IDisposable proporcionan un método Dispose que puede llamarse para liberar recursos en forma explícita. En el capítulo 12, Manejo de excepciones, hablaremos un poco acerca de IDisposable. Puede aprender más acerca de esta interfaz en msdn2.microsoft.com/en-us/library/aax125c9 (inglés), o en msdn2.microsoft.com/es-es/library/system.idisposable(VS.80).aspx (español). El artículo de MSDN <i>Implementar un método Dispose</i> en msdn2.microsoft.com/es-es/library/fs2xkftw(VS.80).aspx habla sobre cómo implementar en forma apropiada esta interfaz en sus clases.
IEnumerator	Se utiliza para iterar a través de los elementos de una colección (como un arreglo), un elemento a la vez. La interfaz IEnumerator contiene el método MoveNext para desplazarse al siguiente elemento en una colección, el método Reset para desplazarse a la posición anterior al primer elemento, y la propiedad Current para devolver el objeto en la ubicación actual. En el capítulo 26, Colecciones, utilizaremos a IEnumerator.

Figura 11.16 | Interfaces comunes de la Biblioteca de clases del .NET Framework. (Parte 2 de 2).



Observación de ingeniería de software 11.9

Use la sobrecarga de operadores cuando ésta haga que la aplicación sea más clara que lograr las mismas operaciones con llamadas explícitas a métodos.

C# le permite sobrecargar la mayoría de los operadores para hacerlos sensibles al contexto en el que se utilicen. Algunos operadores se sobrecargan con frecuencia, en especial varios operadores aritméticos tales como + y -. El trabajo realizado por los operadores sobrecargados también puede realizarse mediante llamadas explícitas a métodos, pero la notación de los operadores es por lo común más natural. Las figuras 11.17 y 11.18 muestran un ejemplo del uso de la sobrecarga de operadores, con una clase *NúmeroComplejo*.

La clase *NúmeroComplejo* (figura 11.17) sobrecarga los operadores de suma (+), resta (-) y multiplicación (*) para permitir a los programas sumar, restar y multiplicar instancias de la clase *NúmeroComplejo* mediante el uso de la notación matemática común. Las líneas 8-9 declaran variables de instancia para las partes real e imaginaria del número complejo.

```

1 // Fig. 11.17: NúmeroComplejo.cs
2 // Clase que sobrecarga operadores para sumar, restar
3 // y multiplicar números complejos.
4 using System;
5
6 public class NúmeroComplejo
7 {
8     private double real; // componente real del número complejo
9     private double imaginario; // componente imaginario del número complejo
10
11    // constructor
12    public NúmeroComplejo( double a, double b )
13    {
14        real = a;
15        imaginario = b;

```

Figura 11.17 | Clase que sobrecarga operadores para sumar, restar y multiplicar números complejos. (Parte 1 de 2).

```

16 } // fin del constructor
17
18 // devuelve representación string de NumeroComplejo
19 public override string ToString()
20 {
21     return string.Format( "{0} {1} {2}i",
22         Real, ( Imaginario < 0 ? "-" : "+" ), Math.Abs( Imaginario ) );
23 } // fin del método ToString
24
25 // propiedad de sólo lectura que obtiene el componente real
26 public double Real
27 {
28     get
29     {
30         return real;
31     } // fin de get
32 } // fin de la propiedad Real
33
34 // propiedad de sólo lectura que obtiene el componente imaginario
35 public double Imaginario
36 {
37     get
38     {
39         return imaginario;
40     } // fin de get
41 } // fin de la propiedad Imaginario
42
43 // sobrecarga el operador de suma
44 public static NumeroComplejo operator+
45     NumeroComplejo x, NumeroComplejo y )
46 {
47     return new NumeroComplejo( x.Real + y.Real,
48         x.Imaginario + y.Imaginario );
49 } // fin del operador +
50
51 // sobrecarga el operador de resta
52 public static NumeroComplejo operator-
53     NumeroComplejo x, NumeroComplejo y )
54 {
55     return new NumeroComplejo( x.Real - y.Real,
56         x.Imaginario - y.Imaginario );
57 } // fin del operador -
58
59 // sobrecarga el operador de multiplicación
60 public static NumeroComplejo operator*
61     NumeroComplejo x, NumeroComplejo y )
62 {
63     return new NumeroComplejo(
64         x.Real * y.Real - x.Imaginario * y.Imaginario,
65         x.Real * y.Imaginario + y.Real * x.Imaginario );
66 } // fin del operador *
67 } // fin de la clase NumeroComplejo

```

Figura 11.17 | Clase que sobrecarga operadores para sumar, restar y multiplicar números complejos. (Parte 2 de 2).

Las líneas 44-49 sobrecargan el operador de suma (+) para realizar la suma de objetos `NumeroComplejo`. La palabra clave **operator**, seguida de un símbolo de operador, indica que un método sobrecarga el operador específico. Los métodos que sobrecargan operadores binarios deben recibir dos argumentos. El primer argumento

es el operando izquierdo, y el segundo argumento es el operando derecho. El operador de suma sobrecargado de la clase `NumeroComplejo` recibe dos referencias `NumeroComplejo` como argumentos y devuelve un `NumeroComplejo` que representa la suma de los argumentos. Observe que este método se marca como `public` y `static`, obligatorio para los operadores sobrecargados. El cuerpo del método (líneas 47-48) realiza la suma y devuelve el resultado como un nuevo `NumeroComplejo`. Observe que no modificamos el contenido de ninguno de los operandos originales que se pasan como los argumentos `x` y `y`. Esto concuerda con nuestro sentido intuitivo de cómo debe comportarse este operador: al sumar dos números no se modifica ninguno de los números originales. Las líneas 52-66 proporcionan operadores sobrecargados similares para restar y multiplicar objetos `NumeroComplejo`.



Observación de ingeniería de software 11.10

Sobrecargue operadores para realizar la misma función o funciones similares en los objetos de clases que los operadores realizan en objetos de tipos simples. Evite el uso no intuitivo de los operadores.



Observación de ingeniería de software 11.11

Al menos un argumento del método de un operador sobrecargado debe ser una referencia a un objeto de la clase en la que se sobrecarga el operador. Esto evita que los programadores cambien la forma en que trabajan los operadores con los tipos simples.

La clase `PruebaComplejo` (figura 11.18) demuestra el uso de los operadores sobrecargados para sumar, restar y multiplicar objetos `NumeroComplejo`. Las líneas 14-27 piden al usuario que introduzca dos números complejos, y después utilizan esta entrada para crear dos objetos `NumeroComplejo` y asignarlos a las variables `x` y `y`.

```

1 // Fig 11.18: PruebaComplejo.cs
2 // Sobre carga de operadores para números complejos.
3 using System;
4
5 public class PruebaComplejo
6 {
7     public static void Main( string[] args )
8     {
9         // declara dos variables para almacenar números complejos
10        // que introduce el usuario
11        NumeroComplejo x, y;
12
13        // pide al usuario que introduzca el primer número complejo
14        Console.WriteLine("Escriba la parte real del número complejo x: ");
15        double parteReal = Convert.ToDouble(Console.ReadLine());
16        Console.WriteLine(
17            "Escriba la parte imaginaria del número complejo x: ");
18        double parteImaginaria = Convert.ToDouble(Console.ReadLine());
19        x = new NumeroComplejo(parteReal, parteImaginaria);
20
21        // pide al usuario que introduzca el segundo número complejo
22        Console.WriteLine("\nEscriba la parte real del número complejo y: ");
23        parteReal = Convert.ToDouble(Console.ReadLine());
24        Console.WriteLine(
25            "Escriba la parte imaginaria del número complejo y: ");
26        parteImaginaria = Convert.ToDouble(Console.ReadLine());
27        y = new NumeroComplejo(parteReal, parteImaginaria);
28
29        // muestra en pantalla los resultados de los cálculos con x y y
30        Console.WriteLine();
31        Console.WriteLine("{0} + {1} = {2}", x, y, (x + y).ToString());

```

Figura 11.18 | Sobre carga de operadores para números complejos. (Parte 1 de 2).

```

32     Console.WriteLine( "{0} - {1} = {2}", x, y, x - y );
33     Console.WriteLine( "{0} * {1} = {2}", x, y, x * y );
34 } // fin del método Main
35 } // fin de la clase PruebaComplejo

```

```

Escriba la parte real del número complejo x: 2
Escriba la parte imaginaria del número complejo x: 4

Escriba la parte real del número complejo y: 4
Escriba la parte imaginaria del número complejo y: -2

(2 + 4i) + (4 - 2i) = (6 + 2i)
(2 + 4i) - (4 - 2i) = (-2 + 6i)
(2 + 4i) * (4 - 2i) = (16 + 12i)

```

Figura 11.18 | Sobre carga de operadores para números complejos. (Parte 2 de 2).

Las líneas 31-33 suman, restan y multiplican x y y con los operadores sobre cargados, y después imprimen los resultados en pantalla. En la línea 31, para realizar la suma utilizamos el operador de suma con los operandos `NumeroComplejo` x y y . Sin la sobre carga de operadores, la expresión $x + y$ no tendría sentido; el compilador no sabría cómo deberían sumarse los dos objetos. Esta expresión tiene sentido aquí debido a que hemos definido el operador de suma para dos objetos `NumeroComplejo` en las líneas 44-49 de la figura 11.17. Cuando se “suman” los dos objetos `NumeroComplejo` en la línea 31 de la figura 11.18, se invoca la declaración de `operador+`, se pasa el operando izquierdo como el primer argumento y el operando derecho como el segundo argumento. Cuando utilizamos los operadores de resta y multiplicación en las líneas 32-33, se invocan de manera similar sus respectivas declaraciones de operadores sobre cargados.

Observe que el resultado de cada cálculo es una referencia a un nuevo objeto `NumeroComplejo`. Cuando se pasa este nuevo objeto al método `WriteLine` de la clase `Console`, se invoca en forma implícita a su método `ToString` (líneas 19-23 de la figura 11.17). No necesitamos asignar un objeto a una variable de tipo de referencia para invocar a su método `ToString`. La línea 31 de la figura 11.18 podría reescribirse para invocar en forma explícita al método `ToString` del objeto creado por el operador de suma sobre cargado, como en:

```
Console.WriteLine( "{0} + {1} = {2}", x, y, ( x + y ).ToString() );
```

11.9 (Opcional) Caso de estudio de ingeniería de software: incorporación de la herencia y el polimorfismo en el sistema ATM

Ahora regresaremos a nuestro diseño del sistema ATM para ver cómo podría beneficiarse de la herencia y el polimorfismo. Para aplicar la herencia, primero buscamos características comunes entre las clases del sistema. Creamos una jerarquía de herencia para modelar las clases similares en una forma elegante y eficiente que nos permita procesar objetos de estas clases mediante el polimorfismo. Despues modificamos nuestro diagrama de clases para incorporar las nuevas relaciones de herencia. Por último, demostramos cómo traducir los aspectos relacionados con la herencia de nuestro diseño actualizado, a código de C#.

En la sección 4.11 nos topamos con el problema de representar una transacción financiera en el sistema. En vez de crear una clase para representar a todos los tipos de transacciones, creamos tres clases distintas de transacciones (`SolicitudSaldo`, `Retiro` y `Deposito`) para representar las transacciones que puede realizar el sistema ATM. El diagrama de clases de la figura 11.19 muestra los atributos y operaciones de estas clases. Observe que tienen un atributo privado (`numeroCuenta`) y una operación pública (`Ejecutar`) en común. Cada clase requiere que el atributo `numeroCuenta` especifique la cuenta a la que se aplica la transacción.

Cada clase contiene la operación `Ejecutar()`, que el ATM invoca para realizar la transacción. Es evidente que `SolicitudSaldo`, `Retiro` y `Deposito` representan *tipos de transacciones*. La figura 11.19 revela las características comunes entre las clases de transacciones, por lo que el uso de la herencia para factorizar las características comunes parece apropiado para diseñar estas clases. Colocamos la funcionalidad común en la clase base `Transaccion` y derivamos las clases `SolicitudSaldo`, `Retiro` y `Deposito` de `Transaccion` (figura 11.20).

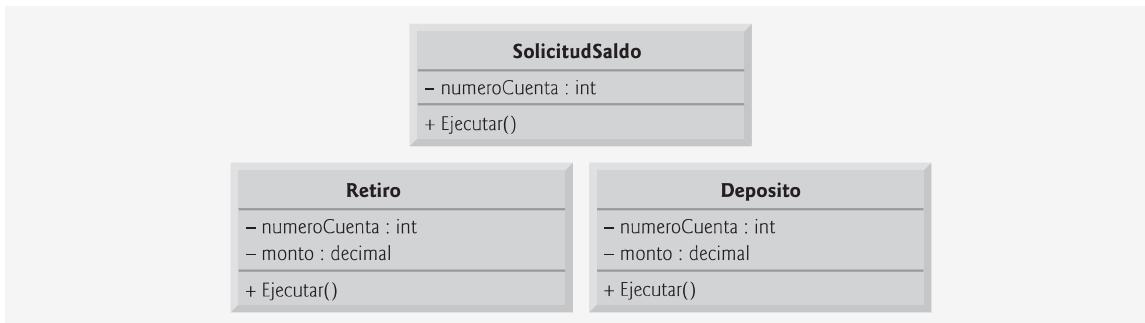


Figura 11.19 | Atributos y operaciones de las clases *SolicitudSaldo*, *Retiro* y *Deposito*.

UML especifica una relación conocida como *generalización* para modelar la herencia. La figura 11.20 es el diagrama de clases que modela la relación de herencia entre la clase base *Transaccion* y sus tres clases derivadas. Las flechas con puntas triangulares huecas indican que las clases *SolicitudSaldo*, *Retiro* y *Deposito* se derivan de la clase *Transaccion* por herencia. Se dice que la clase *Transaccion* es una generalización de sus clases derivadas. Se dice que las clases derivadas son *especializaciones* de la clase *Transaccion*.

Como se muestra en la figura 11.19, las clases *SolicitudSaldo*, *Retiro* y *Deposito* comparten el atributo privado *int numeroCuenta*. Nos gustaría factorizar este atributo común y colocarlo en la clase base *Transaccion*. Sin embargo, recuerde que los atributos privados de una clase base no son accesibles en las clases derivadas. Las clases derivadas de *Transaccion* requieren acceso al atributo *numeroCuenta*, para poder especificar cuál Cuenta se va a procesar en la *BaseDatosBanco*. Como vimos en el capítulo 10, una clase derivada sólo puede tener acceso a los miembros *public*, *protected internal* de su clase base. No obstante, las clases derivadas en este caso no necesitan modificar el atributo *numeroCuenta*; sólo necesitan acceder a su valor. Por esta razón hemos optado por sustituir el atributo privado *numeroCuenta* en nuestro modelo con la propiedad pública de sólo lectura *NumeroCuenta*. Como ésta es una propiedad de sólo lectura, sólo proporciona un descriptor de acceso *get* para acceder al número de cuenta. Cada clase derivada hereda esta propiedad, lo cual le permite acceder a su número de cuenta según lo necesite para ejecutar una transacción. Ya no listamos *numeroCuenta* en el segundo compartimiento de cada clase derivada, ya que las tres clases derivadas heredan la propiedad *NumeroCuenta* de *Transaccion*.

De acuerdo con la figura 11.19, las clases *SolicitudSaldo*, *Retiro* y *Deposito* también comparten la operación *Ejecutar*, por lo que la clase base *Transaccion* debería contener la operación *public Ejecutar()*. Sin embargo, no tiene sentido implementar a *Ejecutar* en la clase *Transaccion*, ya que la funcionalidad que proporciona esta operación depende del tipo específico de la transacción actual. Por lo tanto, declaramos a

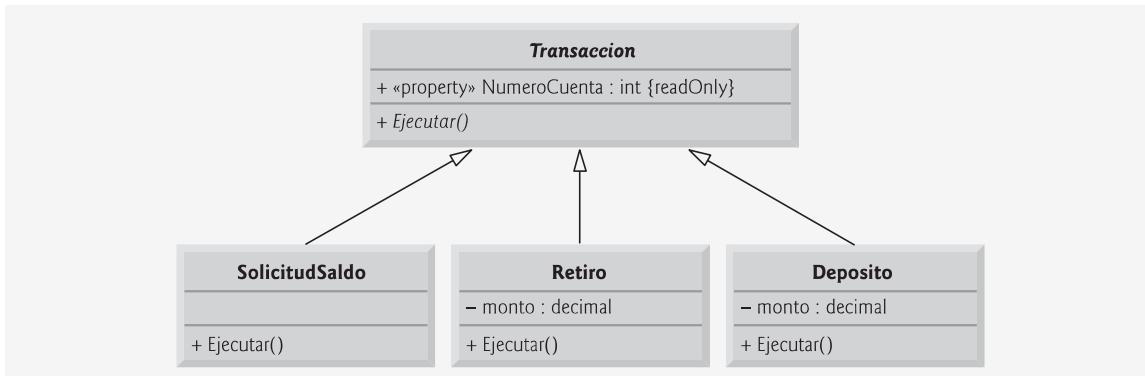


Figura 11.20 | Diagrama de clases que modela la relación de generalización (es decir, herencia) entre la clase base *Transaccion* y sus clases derivadas *SolicitudSaldo*, *Retiro* y *Deposito*.

Ejecutar como una *operación abstracta* en la clase base **Transaccion**; se convertirá en un método **abstract** en la implementación en C#. Esto hace a **Transaccion** una clase abstracta y obliga a cualquier clase derivada de **Transaccion** que deba ser una clase concreta (es decir, **SolicitudSaldo**, **Retiro** y **Deposito**) a implementar la operación **Ejecutar**, para hacer que la clase derivada sea concreta. UML requiere que coloquemos los nombres de clase abstractos y las operaciones abstractas en cursivas. Por ende, en la figura 11.20 **Transaccion** y **Ejecutar** aparecen en cursivas para la clase **Transaccion**; **Ejecutar** no se coloca en cursivas en las clases derivadas **SolicitudSaldo**, **Retiro** y **Deposito**. Cada clase derivada redefine a la operación **Ejecutar** de **Transaccion** con una implementación concreta apropiada. Observe que la figura 11.20 incluye la operación **Ejecutar** en el tercer compartimiento de las clases **SolicitudSaldo**, **Retiro** y **Deposito**, ya que cada clase tiene una implementación concreta distinta de la operación redefinida.

Como aprendió en este capítulo, una clase derivada puede heredar la interfaz y la implementación de una clase base. En comparación con una jerarquía diseñada para la herencia de implementación, una diseñada para la herencia de interfaz tiende a tener su funcionalidad a un nivel más bajo en la jerarquía; una clase base significa una o más operaciones que debe definir cada clase en la jerarquía, pero las clases derivadas individuales proporcionan sus propias implementaciones de la(s) operación(es). La jerarquía de herencia diseñada para el sistema ATM aprovecha este tipo de herencia, la cual proporciona a la clase ATM una manera elegante de ejecutar todas las transacciones “en general” (es decir, en forma polimórfica). Cada clase derivada de **Transaccion** hereda ciertos detalles de implementación (por ejemplo, la propiedad **NumeroCuenta**), pero el principal beneficio de incorporar la herencia en nuestro sistema es que las clases derivadas comparten una interfaz común (por ejemplo, la operación abstracta **Ejecutar**). La clase ATM puede dirigir una referencia **Transaccion** a cualquier transacción, y cuando ATM invoca a la operación **Ejecutar** a través de esta referencia, se ejecuta la versión de **Ejecutar** específica para esa transacción (en forma polimórfica) de manera automática (debido al polimorfismo). Por ejemplo, suponga que un usuario selecciona la opción para solicitar su saldo. La clase ATM dirige una referencia **Transaccion** a un nuevo objeto de la clase **SolicitudSaldo**, que el compilador de C# permite debido a que un objeto **SolicitudSaldo** es un objeto **Transaccion**. Cuando la clase ATM utiliza esta referencia para invocar a **Ejecutar**, se llama a la versión de **Ejecutar** de **SolicitudSaldo** (en forma polimórfica).

Este enfoque polimórfico también facilita la extensibilidad del sistema. Si deseamos crear un nuevo tipo de transacción (por ejemplo, una transferencia de fondos o el pago de un recibo), tan sólo tenemos que crear una clase derivada de **Transaccion** adicional que redefina la operación **Ejecutar** con una versión apropiada para el nuevo tipo de transacción. Sólo tendríamos que realizar pequeñas modificaciones al código del sistema, para permitir que los usuarios seleccionaran el nuevo tipo de transacción del menú principal y para que la clase ATM creara instancias y ejecutara objetos de la nueva clase derivada. La clase ATM podría ejecutar transacciones del nuevo tipo utilizando el código actual, ya que éste ejecuta todas las transacciones de manera idéntica (a través del polimorfismo).

Como aprendió antes en este capítulo, una clase abstracta tal como **Transaccion** es una para la cual el programador nunca tendrá la intención de (y de hecho, no podrá) crear objetos. Una clase abstracta sólo declara los atributos y comportamientos comunes para sus clases derivadas en una jerarquía de herencia. La clase **Transaccion** define el concepto de lo que significa ser una transacción que tiene un número de cuenta y puede ejecutarse. Tal vez usted se pregunte por qué nos tomamos la molestia de incluir la operación abstracta **Ejecutar** en la clase **Transaccion**, si **Ejecutar** carece de una implementación completa. En concepto, incluimos esta operación porque es el comportamiento que define a todas las transacciones: ejecutarse. Técnicamente, debemos incluir la operación **Ejecutar** en la clase base **Transaccion**, de manera que la clase ATM (o cualquier otra clase) pueda invocar a la versión redefinida de esta operación de cada clase derivada en forma polimórfica, a través de una referencia **Transaccion**.

Las clases derivadas **SolicitudSaldo**, **Retiro** y **Deposito** heredan la propiedad **NumeroCuenta** de la clase base **Transaccion**, pero las clases **Retiro** y **Deposito** contienen el atributo adicional **monto** que las diferencia de la clase **SolicitudSaldo**. Las clases **Retiro** y **Deposito** requieren este atributo adicional para almacenar el monto de dinero que el usuario desea retirar o depositar. La clase **SolicitudSaldo** no necesita dicho atributo puesto que sólo requiere un número de cuenta para ejecutarse. Aun cuando dos de las tres clases derivadas de **Transaccion** comparten el atributo **monto**, no lo colocamos en la clase base **Transaccion**; en la clase base sólo colocamos las características comunes para *todas* las clases derivadas, para que éstas no hereden atributos (y operaciones) innecesarios.

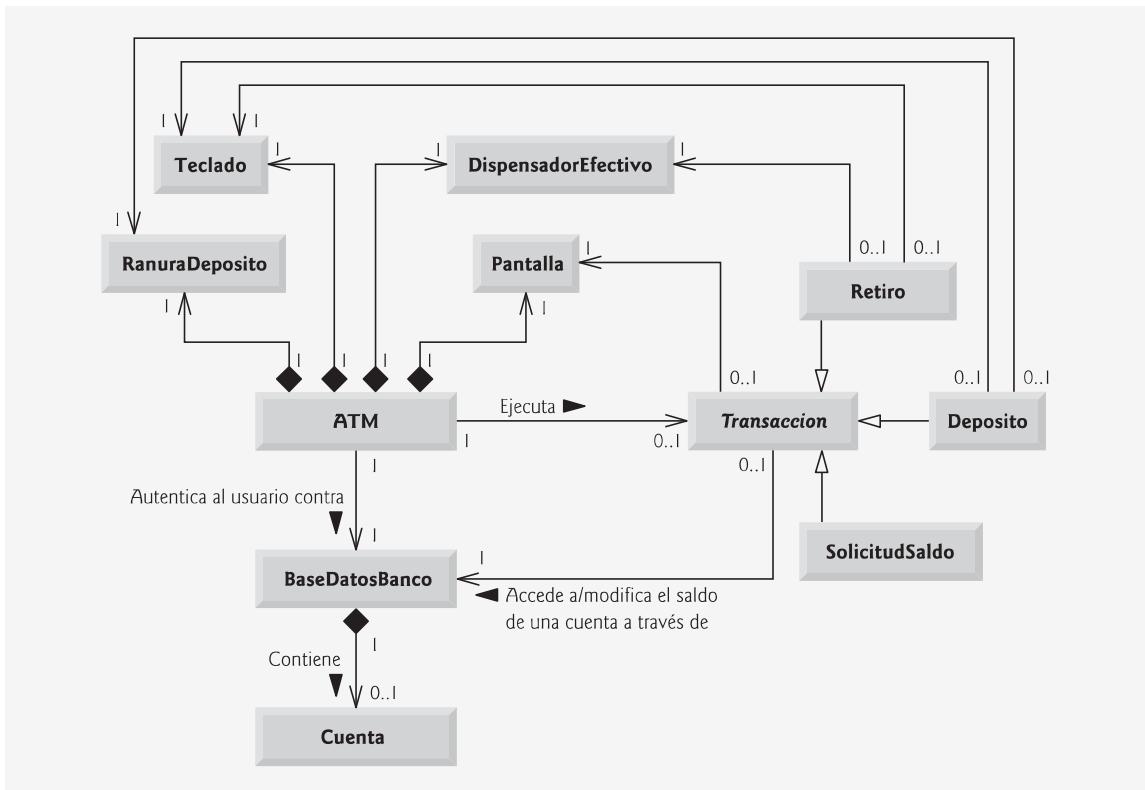


Figura 11.21 | Diagrama de clases del sistema ATM (en el que se incorpora la herencia). Observe que el nombre de la clase abstracta *Transaccion* aparece en cursivas.

La figura 11.21 presenta un diagrama de clases actualizado de nuestro modelo, en el cual se incorpora la herencia y se introduce la clase base abstracta *Transaccion*. Modelamos una asociación entre la clase ATM y la clase *Transaccion* para mostrar que la clase ATM, en cualquier momento dado, está ejecutando una transacción o no lo está (es decir, existen cero o un objetos de tipo *Transaccion* en el sistema, en cualquier momento dado). Como un *Retiro* es un tipo de *Transaccion*, ya no dibujamos una línea de asociación directamente entre la clase ATM y la clase *Retiro*; la clase derivada *Retiro* hereda la asociación de la clase *Transaccion* con la clase ATM. Las clases derivadas *SolicitudSaldo* y *Deposito* también heredan esta asociación, la cual sustituye a las asociaciones que omitimos antes entre las clases *SolicitudSaldo* y *Deposito*, y la clase ATM. Observe de nuevo el uso de las puntas de flecha triangulares sin relleno para indicar las especializaciones (es decir, clases derivadas) de la clase *Transaccion*, como se indica en la figura 11.20.

También agregamos una asociación entre la clase *Transaccion* y la clase *BaseDatosBanco* (figura 11.21). Todos los objetos *Transaccion* requieren una referencia a *BaseDatosBanco*, de manera que puedan acceder y modificar la información de las cuentas. Cada clase derivada de *Transaccion* hereda esta referencia, por lo que ya no tenemos que modelar la asociación entre la clase *Retiro* y *BaseDatosBanco*. Observe que la asociación entre la clase *Transaccion* y *BaseDatosBanco* sustituye a las asociaciones que omitimos antes entre las clases *SolicitudSaldo* y *Deposito*, y *BaseDatosBanco*.

Incluimos una asociación entre la clase *Transaccion* y la clase *Pantalla*, debido a que todos los objetos *Transaccion* muestran los resultados al usuario a través de la *Pantalla*. Cada clase derivada hereda esta asociación. Por lo tanto, ya no incluimos la asociación que modelamos antes entre *Retiro* y *Pantalla*. No obstante, la clase *Retiro* aún participa en las asociaciones con *DispensadorEfectivo* y *Teclado*; estas asociaciones se aplican a la clase derivada *Retiro*, pero no a las clases derivadas *SolicitudSaldo* y *Deposito*, por lo que no movemos estas asociaciones a la clase base *Transaccion*.

Nuestro diagrama de clases que incorpora la herencia (figura 11.21) también modela las clases **Deposito** y **SolicitudSaldo**. Mostramos las asociaciones entre **Deposito** y las clases **RanuraDeposito** y **Teclado**. Observe que la clase **SolicitudSaldo** participa sólo en las asociaciones heredadas de la clase **Transaccion**; una **SolicitudSaldo** interactúa sólo con la **BaseDatosBanco** y la **Pantalla**.

El diagrama de clases de la figura 9.23 muestra los atributos, las propiedades y las operaciones con marcas de visibilidad. Ahora presentamos un diagrama de clases modificado en la figura 11.22, el cual incluye la clase base abstracta **Transaccion**. Este diagrama abreviado no muestra las relaciones de herencia (éstas aparecen en la figura 11.21), sino los atributos y las operaciones después de haber empleado la herencia en nuestro sistema. Observe que el nombre de la clase base abstracta **Transaccion** y el nombre de la operación abstracta **Ejecutar** aparecen en cursivas. Para ahorrar espacio, como hicimos en la figura 5.16, no incluimos los atributos mostrados por las asociaciones en la figura 11.22; sin embargo, los incluimos en la implementación en C# del apéndice J. También omitimos todos los parámetros de las operaciones, como hicimos en la figura 9.23; al incorporar la herencia no se afectan los parámetros que ya estaban modelados en las figuras 7.22-7.24.

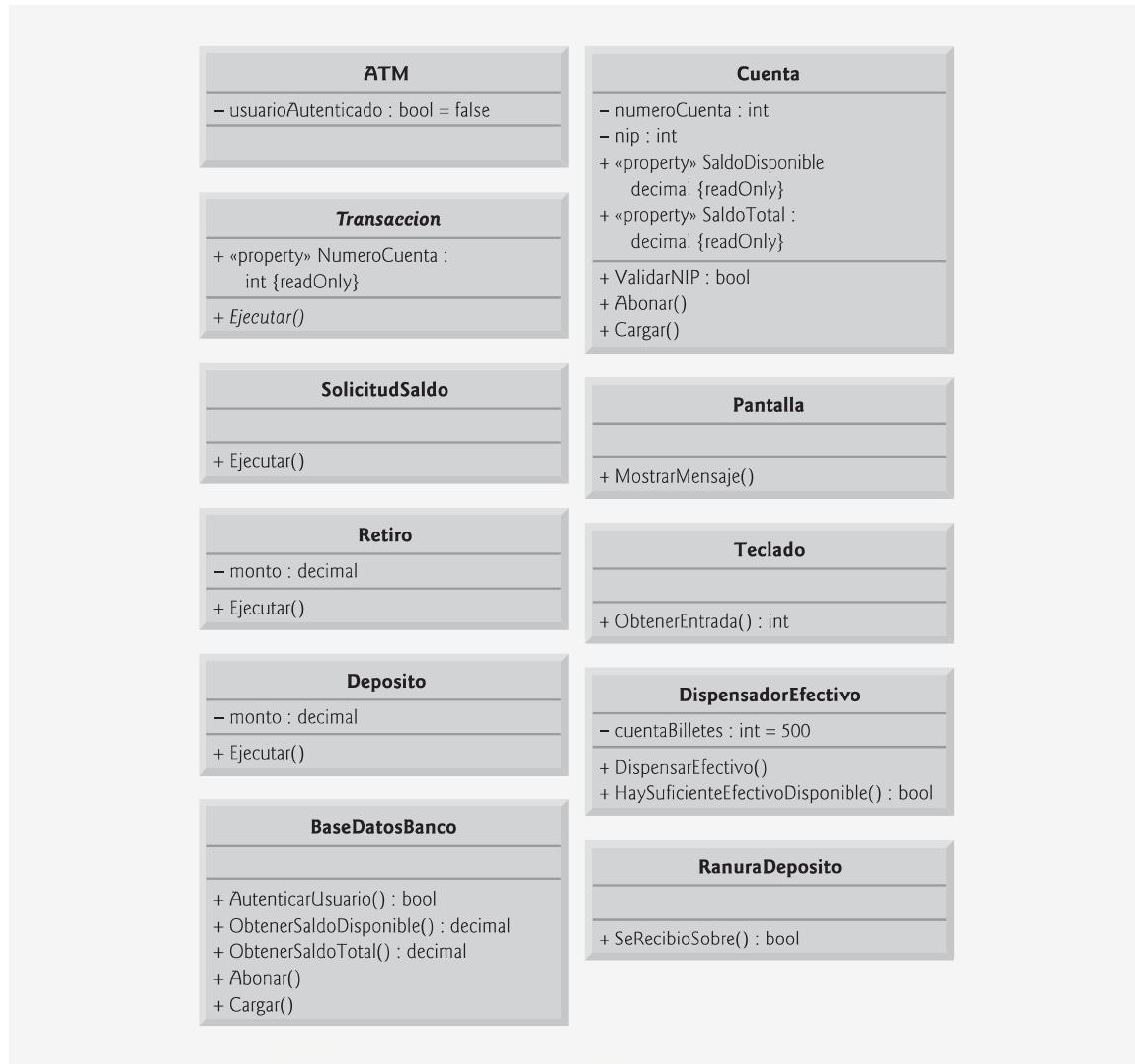


Figura 11.22 | Diagrama de clases después de incorporar la herencia en el sistema.



Observación de ingeniería de software 11.12

Un diagrama de clases completo muestra todas las asociaciones entre clases, junto con todos los atributos y operaciones para cada clase. Cuando el número de atributos, operaciones y asociaciones de las clases es substancial (como en las figuras 11.21 y 11.22), una buena práctica que promueve la legibilidad es dividir esta información entre dos diagramas de clases: uno que se enfoca en las asociaciones y el otro en los atributos y operaciones. Al examinar las clases modeladas en esta forma, es imprescindible considerar ambos diagramas de clases para obtener una idea completa acerca de las clases. Por ejemplo, uno debe referirse a la figura 11.21 para observar la relación de herencia entre `Transaccion` y sus clases derivadas; esa relación se omite en la figura 11.22.

Implementación del diseño del sistema ATM en el que se incorpora la herencia

En la sección 9.17 empezamos a implementar el diseño del sistema ATM en código de C#. Ahora modificaremos nuestra implementación para incorporar la herencia, usando la clase `Retiro` como ejemplo.

1. Si la clase A es una generalización de la clase B, entonces la clase B se deriva (y es una especialización) de la clase A. Por ejemplo, la clase base abstracta `Transaccion` es una generalización de la clase `Retiro`. Por ende, la clase `Retiro` se deriva (y es una especialización) de la clase `Transaccion`. La figura 11.23 contiene la estructura de la clase `Retiro`, en la que la definición de la clase indica la relación de herencia entre `Retiro` y `Transaccion` (línea 3).
2. Si la clase A es una clase abstracta y la clase B se deriva de la clase A, entonces la clase B debe implementar las operaciones abstractas de la clase A, si la clase B va a ser una clase concreta. Por ejemplo, la clase `Transaccion` contiene la operación abstracta `Ejecutar`, por lo que la clase `Retiro` debe implementar esta operación si queremos crear instancias de objetos `Retiro`. La figura 11.24 contiene las porciones del código en C# para la clase `Retiro` que pueden inferirse de las figuras 11.21 y 11.22. La clase `Retiro` hereda la propiedad `NumeroCuenta` de la clase base `Transaccion`, por lo que `Retiro` no declara esta propiedad. La clase `Retiro` también hereda referencias a las clases `Pantalla` y `BaseDatosBanco` de la clase `Transaccion`, por lo que no incluimos estas referencias en nuestro código. La figura 11.22 especifica el atributo `monto` y la operación `Ejecutar` para la clase `Retiro`. La línea 6 de la figura 11.24 declara una variable de instancia para el atributo `monto`. Las líneas 17-20 declaran la estructura de un método para la operación `Ejecutar`. Recuerde que la clase derivada `Retiro` debe proporcionar una implementación concreta del método abstracto `Ejecutar` de la clase base `Transaccion`. Las referencias `teclado` y `dispensadorEfectivo` (líneas 7-8) son variables de instancia, cuya necesidad es aparente debido a las asociaciones de la clase `Retiro` en la figura 11.21; en la implementación en C# de esta clase en el apéndice J, un constructor inicializa estas referencias a objetos reales.

```

1 // Fig 11.23: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM.
3 public class Retiro : Transaccion
4 {
5     // código para los miembros de la clase Retiro
6 } // fin de la clase Retiro

```

Figura 11.23 | Código en C# para la estructura de la clase `Retiro`.

```

1 // Fig 11.24: Retiro.cs
2 // La clase Retiro representa una transacción de retiro del ATM.
3 public class Retiro : Transaccion
4 {
5     // atributos
6     private decimal monto; // monto a retirar
7     private Teclado teclado; // referencia al teclado

```

Figura 11.24 | Código en C# para la clase `Retiro`, con base en las figuras 11.21 y 11.22. (Parte I de 2).

```

8  private DispensadorEfectivo dispensadorEfectivo; // referencia al dispensador de
   efectivo
9
10 // constructor sin parámetros
11 public Retiro()
12 {
13     // código del cuerpo del constructor
14 } // fin del constructor
15
16 // método que redefine a Ejecutar
17 public override void Ejecutar()
18 {
19     // código del cuerpo del método Ejecutar
20 } // fin del método Ejecutar
21 } // fin de la clase Retiro

```

Figura 11.24 | Código en C# para la clase *Retiro*, con base en las figuras 11.21 y 11.22. (Parte 2 de 2).

En la sección J.2 de la implementación del ATM hablaremos sobre el procesamiento polimórfico de objetos *Transaccion*. La clase ATM realiza la llamada polimórfica al método *Ejecutar* en la línea 99 de la figura J.1.

Conclusión del caso de estudio sobre el ATM

Esto concluye nuestro diseño orientado a objetos del sistema ATM. En el apéndice J aparece una implementación en C# completa del sistema ATM, en 655 líneas de código. Esta implementación funcional utiliza la mayor parte de los conceptos de programación orientada a objetos que hemos cubierto hasta este punto en el libro, incluyendo: clases, objetos, encapsulamiento, visibilidad, composición, herencia y polimorfismo. El código contiene abundantes comentarios y se conforma a las prácticas de codificación que usted ha aprendido hasta ahora. Dominar este código será una maravillosa experiencia culminante para usted, después de haber estudiado las nueve secciones del Caso de estudio de ingeniería de software en los capítulos 1, 3 al 9 y 11.

Ejercicios de autoevaluación del Caso de estudio de ingeniería de software

- 11.1 UML utiliza una flecha con una _____ para indicar una relación de generalización.
 - a) punta con relleno sólido
 - b) punta triangular sin relleno
 - c) punta hueca en forma de diamante
 - d) punta lineal
- 11.2 Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: UML requiere que subramos los nombres de las clases abstractas y los nombres de las operaciones abstractas.
- 11.3 Escriba código en C# para empezar a implementar el diseño para la clase *Transaccion* que se especifica en las figuras 11.21 y 11.22. Asegúrese de incluir las referencias **private** basadas en las asociaciones de la clase *Transaccion*. Asegúrese también de incluir las propiedades con los descriptores de acceso **public get** para cualquiera de las variables de instancia **private** a las que deban acceder las clases derivadas, para realizar sus tareas.

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

- 11.1 b.
- 11.2 Falso. UML requiere que se escriban los nombres de las clases y de las operaciones en cursivas.
- 11.3 El diseño para la clase *Transaccion* produce el código de la figura 11.25. En la implementación en el apéndice J, un constructor inicializa las variables de instancia **private PantallaUsuario** y **baseDatos** para objetos reales, y las propiedades de sólo lectura **PantallaUsuario** y **BaseDatos** acceden a estas variables de instancia. Estas propiedades permiten que las clases derivadas de *Transaccion* accedan a la pantalla del ATM e interactúen con la base de datos del banco. Observe que elegimos los nombres de las propiedades **PantallaUsuario** y **BaseDatos** por cuestión de claridad; queremos evitar nombres de propiedades que sean iguales que los nombres de las clases **Pantalla** y **BaseDatosBanco**, lo cual puede ser confuso.

```

1 // Fig 11.25: Transaccion.cs
2 // La clase base abstracta Transaccion representa una transacción del ATM.
3 public abstract class Transaccion
4 {
5     private int numeroCuenta; // indica la cuenta involucrada
6     private Pantalla pantallaUsuario; // pantalla del ATM
7     private BaseDatosBanco baseDatos; // base de datos de información de las cuentas
8
9     // constructor sin parámetros
10    public Transaccion()
11    {
12        // código del cuerpo del constructor
13    } // fin del constructor
14
15    // propiedad de sólo lectura que obtiene el número de cuenta
16    public int NumeroCuenta
17    {
18        get
19        {
20            return numeroCuenta;
21        } // fin de get
22    } // fin de la propiedad NumeroCuenta
23
24    // propiedad de sólo lectura que obtiene la referencia a la pantalla
25    public Pantalla pantallaUsuario
26    {
27        get
28        {
29            return pantallaUsuario;
30        } // fin de get
31    } // fin de la propiedad pantallaUsuario
32
33    // propiedad de sólo lectura que obtiene la referencia a la base de datos del banco
34    public BaseDatosBanco BaseDatos
35    {
36        get
37        {
38            return baseDatos;
39        } // fin de get
40    } // fin de la propiedad BaseDatos
41
42    // realiza la transacción (cada clase derivada redefine este método)
43    public abstract void Ejecutar();
44 } // fin de la clase Transaccion

```

Figura 11.25 | Código en C# para la clase Transaccion, con base en las figuras 11.21 y 11.22.

11.10 Conclusión

En este capítulo se introdujo el polimorfismo: la habilidad de procesar objetos que comparten la misma clase base en una jerarquía de clases, como si todos fueran objetos de la clase base. En este capítulo hablamos sobre cómo el polimorfismo facilita la extensibilidad y manejabilidad de los sistemas, y después demostramos cómo utilizar métodos redefinidos para llevar a cabo el comportamiento polimórfico. Presentamos la noción de una clase abstracta, la cual nos proporciona una clase base apropiada, a partir de la cual otras clases pueden heredar. Aprendió que una clase abstracta puede declarar métodos abstractos que debe implementar cada clase derivada para convertirse en clase concreta, y que una aplicación puede utilizar variables de una clase abstracta para invocar implementaciones de clases derivadas de los métodos abstractos en forma polimórfica. También aprendió a determinar el tipo de un objeto en tiempo de ejecución. Le mostramos cómo crear métodos y clases sealed. Hablamos

también sobre la declaración e implementación de una interfaz, como otra manera de lograr el comportamiento polimórfico, a menudo entre objetos de distintas clases. Por último, aprendió a definir el comportamiento de los operadores integrados en objetos de sus propias clases, mediante la sobrecarga de operadores.

Ahora deberá estar familiarizado con las clases, los objetos, el encapsulamiento, la herencia, las interfaces y el polimorfismo: los aspectos más esenciales de la programación orientada a objetos. En el siguiente capítulo analizaremos con más detalle cómo utilizar el manejo de excepciones para lidiar con los errores en tiempo de ejecución.