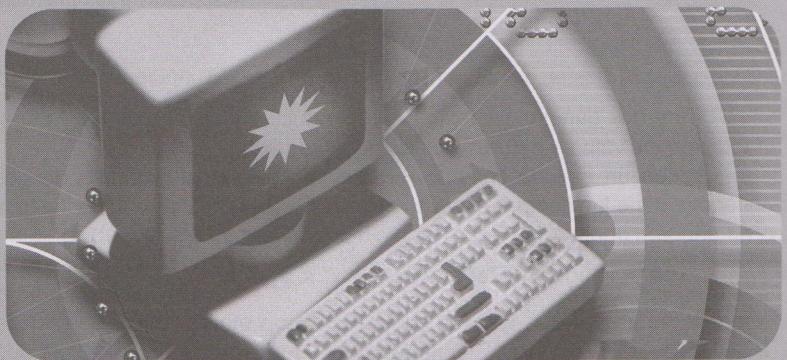


CAPÍTULO 10



Herencia

En este capítulo conoceremos cómo:

- Crear una nueva clase a partir de una clase existente mediante la herencia.
- Aclarar variables y métodos como **protected** y en qué situaciones utilizarlos.
- Utilizar la redefinición y en qué situaciones aplicarla.
- Dibujar un diagrama de clases que describa la herencia.
- Utilizar la palabra clave **super**.
- Escribir constructores para las subclases.
- Emplear la palabra clave **final**.
- Utilizar clases abstractas y la palabra clave **abstract**.

● Introducción

Los programas se crean a partir de objetos, los cuales son instancias de clases. Algunas clases están en la biblioteca de Java y otras las escribe el programador. Cuando empezamos a escribir un nuevo programa, buscamos clases útiles en la biblioteca y vemos las clases que hemos escrito en el pasado. Esta metodología OO para la programación significa que en vez de empezar los programas desde cero, nos basamos en el trabajo anterior. Es común encontrar una clase que parezca útil y que haga casi todo lo que necesitamos, pero que no sea exactamente lo que queremos. La herencia constituye una manera de resolver este problema. Con la herencia podemos usar una clase existente como la base para crear una clase modificada.

Considere la siguiente analogía. Suponga que desea comprar un automóvil nuevo y va a una agencia en donde hay una variedad de automóviles producidos en masa. A usted le gusta uno en especial, pero no tiene esa característica especial que está buscando. Al igual que la descripción de

una clase, el automóvil se fabricó a partir de planos que describen a muchos automóviles idénticos. Si estuviera disponible la herencia, usted podría especificar un automóvil que tuviera todas las características del automóvil producido en masa, pero con las características o cambios adicionales que usted requiere.

Cómo usar la herencia

Vamos a empezar con una clase similar a la que ya hemos usado varias veces en el libro. Es una clase para representar a una esfera. Una esfera tiene un radio y una posición en el espacio. Al mostrar una esfera en la pantalla, debe aparecer como un círculo (el método para mostrar una esfera simplemente invoca al método de biblioteca `drawOval`). El diámetro de la esfera está fijo en 20 píxeles. Sólo hemos modelado las coordenadas *x* y *y* de una esfera (*y* no la coordenada *z*) debido a que vamos a mostrar una representación bidimensional en la pantalla.

He aquí la descripción para la clase `Esfera`:

```
import java.awt.*;
public class Esfera {
    protected int x = 100, y = 100;
    public void setX(int nuevaX) {
        x = nuevaX;
    }
    public void setY(int nuevaY) {
        y = nuevaY;
    }
    public void mostrar(Graphics papel) {
        papel.drawOval(x, y, 20, 20);
    }
}
```

Cabe mencionar que hay varios elementos nuevos en este programa, incluyendo la palabra clave `protected`. Esto se debe a que escribimos la clase de tal forma que se pueda utilizar para la herencia. En el transcurso de este capítulo veremos lo que significan estos nuevos elementos.

Vamos a suponer que alguien escribió y probó esta clase, y la puso a disposición de otros para que la utilicen. Ahora vamos a escribir un nuevo programa y necesitamos una clase muy parecida a ésta, sólo que para describir burbujas. Esta nueva clase llamada `Burbuja` nos permitirá hacer cosas adicionales: modificar el tamaño de una burbuja y moverla en sentido vertical. La limitación de la clase `Esfera` es que describe objetos que no se mueven y cuyo tamaño no se puede cambiar. Primero necesitamos un método adicional que nos permita establecer un nuevo valor para el radio de la burbuja. Podemos hacer esto sin alterar la clase existente; para ello debemos escribir una clase diferente que utilice el código que ya se encuentra en la clase `Esfera`. Decimos que la nueva clase hereda las variables y los métodos de la clase anterior. La nueva clase se convierte en una subclase de la anterior. A la clase anterior se le llama la superclase de la nueva clase. He aquí cómo podemos escribir la nueva clase:

```
import java.awt.*;
public class Burbuja extends Esfera {
    protected int radio = 10;
    public void setTamaño(int tamaño) {
        radio = tamaño;
    }
    public void mostrar(Graphics papel) {
        papel.drawOval(x, y, 2 * radio, 2 * radio);
    }
}
```

Esta nueva clase tiene el nombre **Burbuja**. La palabra clave **extends** y la mención de la clase **Esfera** indican que **Burbuja** hereda de la clase **Esfera**, o decimos que **Burbuja** es una subclase de **Esfera**. Esto significa que **Burbuja** hereda todos los elementos que no estén descritos como **private** dentro de la clase **Esfera**. En las siguientes secciones exploraremos las otras características de esta clase.

● **protected**

Cuando usamos la herencia, **private** es un término demasiado privado y **public** es demasiado público. Si una clase necesita dar a sus subclases acceso a ciertas variables o métodos específicos, pero debe evitar que otras clases accedan a éstos, puede etiquetarlos como **protected**. En la analogía de una familia, una madre permite a sus descendientes utilizar las llaves de su automóvil, pero a nadie más.

Volviendo a la clase **Esfera**, necesitamos variables para describir las coordenadas. Podríamos escribir lo siguiente:

```
private int x, y;
```

Ésta es una decisión acertada, pero debe haber una mejor idea. Podría darse el caso de que en el futuro alguien escribiera una clase que heredara de esta clase y proporcionara un método adicional para mover una esfera. Este método necesitaría acceso a las variables **x** y **y**, que por desgracia son inaccesibles ya que se etiquetaron como **private**. Por lo tanto, para anticiparnos a este posible uso en el futuro, podríamos decidir etiquetarlas mejor como **protected**:

```
protected int x, y;
```

Ahora esta declaración protege a las variables de manera que otras clases no puedan hacer mal uso de ellas, pero permite el acceso a ciertas clases privilegiadas: las subclases. El mismo principio se aplica a los métodos.

Suponga que declaramos a **x** y **y** **private**, como lo teníamos planeado en un principio. La consecuencia es que hubiera sido imposible reutilizar la clase mediante la herencia. La única opción sería editar la clase y reemplazar la descripción **private** por **protected** para esos elementos específicos. Pero esto viola uno de los principios de la POO: nunca modificar una clase existente que haya sido probada y utilizada. Por ende, al escribir una clase debemos esforzarnos para tener en cuenta los posibles usos de la clase en un futuro. El programador que escribe una clase siempre lo hace con la

esperanza de que alguien la reutilice y la extienda. El uso cuidadoso de **protected** en vez de **public** o **private** nos puede ayudar a hacer que una clase sea más atractiva para la herencia. Éste es otro de los principios de la POO.

● Reglas de alcance

En resumen, los cuatro niveles de accesibilidad (reglas de alcance) de una variable o un método en una clase son:

1. **public** – se puede utilizar en cualquier parte. Como regla, cualquier método que ofrezca un servicio a los usuarios de una clase se debe etiquetar como **public**.
2. **protected** – se puede utilizar dentro de esta clase y desde cualquier subclase.
3. **private** – sólo se puede utilizar dentro de esta clase. Por lo general, las variables de instancia se deben declarar como **private** y algunas veces como **protected**.
4. Las variables locales (que se declaran dentro de un método) nunca podrán utilizarse fuera del método específico.

Entonces una clase puede tener un acceso adecuado, pero controlado, a su superclase inmediata y a las superclases que estén arriba de ésta en la jerarquía de clases, como si las clases formaran parte de la clase en sí. Si recurrimos a la analogía de la familia, es como poder gastar libremente el dinero de nuestra madre o de cualquiera de sus ancestros, siempre y cuando hayan puesto su dinero en una cuenta etiquetada como **public** o **protected**. Las personas fuera de la familia sólo pueden acceder al dinero **public**.

● Elementos adicionales

Una forma importante de construir una nueva clase a partir de otra es mediante la inclusión de variables y métodos adicionales.

Podemos ver que la clase **Burbuja** anterior declara una variable y un método adicionales:

```
protected int radio = 10;
public void setTamaño(int tamaño) {
    radio = tamaño;
}
```

La nueva variable es **radio**, adicional a las variables existentes (**x** y **y**) en **Esfera**. Por lo tanto, se extiende el número de variables.

La nueva clase también tiene el método **setTamaño** además de los que contiene **Esfera**.

PRÁCTICA DE AUTOEVALUACIÓN

- 10.1** Un objeto pelota es como un objeto **Esfera**, sólo que con las características adicionales de poder moverse a la izquierda y a la derecha. Escriba una clase llamada **Pelota** que herede la clase **Esfera** pero proporcione los métodos adicionales **moverIzquierda** y **moverDerecha**.

● Redefinición (o sobreescritura)

Otra característica de la nueva clase **Burbuja** es una nueva versión del método **mostrar**:

```
public void mostrar(Graphics papel) {  
    papel.drawOval(x, y, 2 * radio, 2 * radio);  
}
```

Esto es necesario debido a que la nueva clase tiene un radio que se puede modificar, mientras que en la clase **Esfera** el radio estaba fijo. Esta nueva versión de **mostrar** en **Burbuja** sustituye a la versión de la clase **Esfera**. Decimos que la nueva versión *redefine* a la versión anterior.

No confunda la redefinición con la sobrecarga, que vimos en el capítulo 5 sobre los métodos:

- Sobrecargar significa escribir un método (en la misma clase) que tenga el mismo nombre, pero una lista de parámetros distinta.
- Redefinir significa escribir un método en una subclase que tenga el mismo nombre y los mismos parámetros que en la superclase.

En resumen, en la clase que hereda hemos:

- Creado una variable adicional.
- Creado un método adicional.
- Redefinido un método (proporcionamos un método que se va a utilizar en vez del método que ya existía).

Ahora veamos un resumen de lo que hemos logrado. Teníamos una clase llamada **Esfera**. Requeríamos una nueva clase llamada **Burbuja** que fuera similar a **Esfera**, pero necesitábamos herramientas adicionales. Por ende, para crear la nueva clase extendimos las herramientas de la clase que ya teníamos. Hemos explotado al máximo las características comunes entre las dos clases, y hemos evitado volver a escribir piezas del programa que ya existían. Ambas clases que escribimos (**Esfera** y **Burbuja**) están disponibles para que las utilicemos.

Como analogía con las familias humanas, la herencia significa que podemos gastar nuestro propio dinero y también el de nuestra madre.

Cabe mencionar que técnicamente es posible redefinir variables: declarar variables en una subclase que redefinan a las variables de la superclase. No hablaremos más sobre esto por dos buenas razones: la primera, nunca tendremos la necesidad de hacer esto; y la segunda, es muy mala práctica. Al crear una subclase a partir de una clase (heredar de ella), sólo tendremos que:

- Agregar métodos adicionales.
- Agregar variables adicionales.
- Redefinir métodos.

● Diagramas de clases

Una buena forma de visualizar la herencia es mediante un diagrama de clases, como se muestra en la figura 10.1. Esta figura nos muestra que **Burbuja** es una subclase de **Esfera**, la cual a su vez es una subclase de **Object**. Cada clase se muestra como un rectángulo. Una línea entre las clases muestra una relación de herencia. La flecha apunta de la subclase a la superclase.

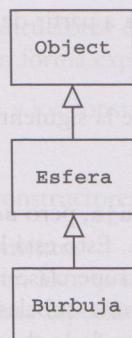


Figura 10.1 Diagrama de clases para las clases **Esfera** y **Burbuja**.

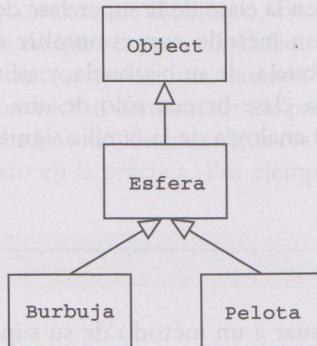


Figura 10.2 Diagrama de clases que muestra una estructura tipo árbol.

Las clases de la biblioteca o las que escribe el programador encajan dentro de una jerarquía de clases. Si usted escribe una clase que empiece con el siguiente encabezado:

```
public class Esfera
```

que no tiene una superclase explícita, entonces es implícitamente una subclase de la clase **Object**. Por lo tanto, toda clase es implícita o explícitamente una subclase de **Object**.

La figura 10.2 muestra otro diagrama de clases en el que otra clase llamada **Pelota** también es una subclase de **Esfera**. Ahora el diagrama es una estructura tipo árbol, en la cual la raíz del árbol, **Object**, está en la parte superior. En general, un diagrama de clases es un árbol, parecido a un árbol genealógico, con la diferencia de que sólo aparece un parente.

● La herencia en acción

Al igual que la clase **Burbuja**, es común que una clase tenga una superclase, la que a su vez tiene una superclase y así sucesivamente, hasta llegar a la parte superior del árbol de herencia. No sólo se heredan los elementos **public** y **protected** de la superclase inmediata, sino también todas las variables y los métodos **public** y **protected** de todas las superclases en el árbol de herencia. Esto es similar al árbol genealógico: usted hereda de su madre, su abuela y así de manera sucesiva.

Suponga que creamos un objeto **burbuja** a partir de la clase **Burbuja**:

```
Burbuja burbuja = new Burbuja();
```

¿Qué ocurre si utilizamos el método **setX** de la siguiente manera?

```
burbuja.setX(200);
```

Aquí **burbuja** es un objeto de la clase **Burbuja**, pero **setX** no es un método de **Burbuja**, sino un método de una clase distinta llamada **Esfera**. Esto está bien, ya que todos los métodos etiquetados como **public** (y **protected**) dentro de la superclase inmediata (y de todas las superclases en la jerarquía de clases) están disponibles para una subclase. Y como **Burbuja** es una subclase de **Esfera**, **setX** está disponible para los objetos de la clase **Burbuja**.

La regla es que al utilizar un método, el sistema de Java primero busca en la clase del objeto para tratar de encontrar el método. Si no lo encuentra ahí, busca en la clase de la superclase inmediata. Si no lo puede encontrar ahí, busca en la clase de la superclase de la superclase, y así recorre toda la jerarquía de clases hasta encontrar un método con el nombre requerido. En la analogía familiar, usted hereda implícitamente de su abuela, de su bisabuela, y así sucesivamente.

El lenguaje Java permite que una clase herede sólo de una superclase inmediata. A esto se le conoce como herencia simple. En la analogía de la familia significa que usted puede heredar de su madre, pero no de su padre.

● **super**

Algunas veces una clase necesita llamar a un método de su superclase inmediata, o de alguna de las clases del nivel superior en el árbol. No hay problema con esto, ya que los métodos de todas las clases en los niveles superiores del árbol de herencia están disponibles, siempre y cuando estén etiquetados como **public** o **protected**. El único problema que puede surgir es cuando el método deseado de la superclase tiene el mismo nombre que un método de la clase actual (cuando se utiliza la redefinición o sobreescritura). Para corregir este problema hay que anteponer al nombre del método la palabra clave **super**. Por ejemplo, para llamar al método **mostrar** de una superclase use lo siguiente:

```
super.mostrar(papel);
```

En general esto es más ordenado y corto que duplicar instrucciones, y nos puede ayudar a que un programa sea más conciso al utilizar al máximo los métodos existentes.

● **Constructores**

En el capítulo 9 vimos sobre los constructores, a la hora de escribir clases. Los constructores nos permiten pasar parámetros a un objeto cuando lo creamos mediante la palabra reservada **new**. Un constructor es un método con el mismo nombre que la clase. Recuerde que:

- Si escribe una clase sin constructores, Java supone que sólo hay un constructor (con cero parámetros).

- Si escribe una clase con uno o más constructores con parámetros y también necesita un constructor sin parámetros, debe escribirlo en forma explícita.

Hay dos reglas relacionadas con la herencia y los constructores:

- Los constructores no se heredan.
- Una subclase debe llamar a uno de los constructores de la superclase.

Veremos cada una de estas reglas en su momento.

Los métodos constructores no se heredan. Ésta es una regla razonable; dice que un constructor está asociado con la inicialización de una clase específica y no de las subclases. Pero significa que si usted necesita uno o más constructores (como se da comúnmente el caso) en una subclase, tiene que escribirlos en forma explícita.

El constructor de una subclase debe llamar a uno de los constructores de la superclase. Además, ésta debe ser la primera acción del constructor. De nuevo, esto es razonable: la superclase se debe inicializar en forma apropiada y antes de que se creen e inicialicen nuevos elementos en la subclase. Si la primera instrucción en un constructor no es una llamada a un constructor de la superclase, entonces Java llama de manera automática al constructor sin parámetros de la superclase. Por ende, Java obliga al programador a llamar a uno de los constructores de la superclase.

Ahora veamos cómo funciona esto en la práctica. Por ejemplo, suponga que tenemos una clase con dos constructores:

```
import java.awt.*;  
public class Globo {  
    protected int x, y, radio;  
    public Globo() {  
        x = 10;  
        y = 10;  
        radio = 20;  
    }  
    public Globo(int xInicial, int yInicial,  
               int radioInicial) {  
        x = xInicial;  
        y = yInicial;  
        radio = radioInicial;  
    }  
    // resto de la clase  
}
```

Si ahora escribimos una nueva clase llamada **GloboDiferente** que herede de la clase **Globo**, las opciones son:

1. No escribir un constructor en la subclase. Java supondrá que hay un constructor sin parámetros.

2. Escribir uno o más constructores sin llamadas a los constructores de la superclase.
3. Escribir uno o más constructores que llamen a un constructor apropiado de la superclase, utilizando **super**.

Para los programadores principiantes (y tal vez para los expertos también) probablemente sea mejor dejar las cosas en claro y escribir de manera explícita una llamada a un constructor de la superclase. Haremos esto en los siguientes ejemplos.

Veamos a continuación una subclase de **Globo** con un constructor que invoca al constructor sin parámetros de la superclase, mediante **super**:

```
public class GloboDiferente extends Globo {
    public GloboDiferente(int xInicial, int yInicial) {
        super();
        x = xInicial;
        y = yInicial;
        radio = 20;
    }
    // resto de la clase
}
```

y he aquí una subclase con un constructor que llama de manera explícita al segundo constructor de la superclase, de nuevo mediante **super**:

```
public class GloboModificado extends Globo {
    public GloboModificado(int xInicial, int yInicial,
                           int radioInicial) {
        super(xInicial, yInicial, radioInicial);
    }
    // resto de la clase
}
```

PRÁCTICA DE AUTOEVALUACIÓN

- 10.2** Una esfera coloreada es como una esfera, sólo que de cierto color. Escriba una nueva clase llamada **EsferaColoreada** que extienda la clase **Esfera** para proporcionar un color que se pueda establecer al momento de crear el globo. Para ello utilice un método constructor, de tal forma que su clase permita escribir lo siguiente:

```
EsferaColoreada esferaColoreada =
    new EsferaColoreada(Color.red);
```

PRÁCTICA DE AUTOEVALUACIÓN

10.3 ¿Cuál es el problema con la subclase en el siguiente código?

```
public class CuentaBancaria {
    protected int depósito;
    CuentaBancaria(int depósitoInicial) {
        // resto del constructor
    }
    // resto de la clase
}

public class CuentaMejorada extends CuentaBancaria {
    public CuentaMejorada() {
        depósito = 1000;
    }
    // resto de la clase
}
```

● final

Los procesos de heredar y redefinir, se enfocan en cambiar el comportamiento de las clases y los objetos. La herencia es muy poderosa, pero algunas veces es reconfortante que algunas cosas estén fijas y no se puedan modificar. Por ejemplo, es bueno saber exactamente qué es lo que hace `sqrt`, qué es lo que hace `drawLine`, etcétera. En la POO siempre existe el peligro de que alguien extienda las clases a las que éstas pertenecen y en consecuencia cambie lo que hacen. Esto podría ser por error o en un intento inadvertido de ser útil. Para evitar esto el programador puede describir un método como `final`. Esto significa que no se puede redefinir. La mayoría de los métodos de la biblioteca se describen como `final`. Esto significa que cada vez que los use, puede estar completamente seguro de lo que hacen.

Como hemos visto desde los primeros capítulos del libro, las variables también se pueden declarar como `final`. Esto significa también que no podemos cambiar sus valores. Son constantes. Por ejemplo:

```
final double cmPerInch = 2.54;
```

declara una variable cuyo valor no se puede alterar. Así, el prefijo `final` tiene el mismo significado sin importar que se utilice en una variable o en un método.

Podemos describir toda una clase como `final`, lo cual significa que no se pueden crear subclases a partir de ella. Además, todos sus métodos serán implícitamente `final`.

Hacer una clase o un método `final` es una decisión seria, ya que impide la herencia: una de las herramientas avanzadas de la POO.

● Clases abstractas

Considere un programa que mantiene formas gráficas de todos tipos y tamaños: círculos, rectángulos, cuadrados, triángulos, etc. Estas distintas formas, similares a las clases que ya hemos visto en este capítulo, tienen información en común: su posición, color y tamaño. Vamos a declarar una superclase llamada **Forma** que describa los datos comunes. Cada clase individual hereda esta información común. He aquí la forma de describir esta superclase común:

```
import java.awt.*;
public abstract class Forma {
    protected int x, y;
    protected int tamaño;
    public void moverDerecha() {
        x = x + 10;
    }
    public abstract void mostrar(Graphics papel);
}
```

La clase para describir círculos hereda de la clase **Forma**, como se muestra a continuación:

```
import java.awt.*;
public class Círculo extends Forma {
    public void mostrar(Graphics papel) {
        papel.drawOval(x, y, tamaño, tamaño);
    }
}
```

Por lo tanto, al escribir la clase **Círculo** hemos aprovechado las herramientas que proporciona la clase **Forma**.

Es inútil tratar de crear un objeto a partir de la clase **Forma**, ya que está incompleta. Ésta es la razón por la cual se incluye la palabra clave **abstract** en el encabezado de la clase **Forma**; el compilador impedirá cualquier intento por crear una instancia de esta clase. Se proporciona el método **moverDerecha** y está completo, y es heredado por cualquier subclase. Pero el método **mostrar** es tan sólo un encabezado (sin cuerpo). Se describe con la palabra clave **abstract** para indicar que cualquier subclase debe proveer una implementación de este método. La clase **Forma** se denomina clase abstracta debido a que no existe como una clase completa, sino que se proporciona simplemente para usarse en la herencia.

Hay una regla razonable que establece que si una clase contiene métodos que sean **abstract**, la misma clase se debe etiquetar también como **abstract**.

Las clases abstractas nos permiten aprovechar las características comunes de las clases. Al declarar una clase como **abstract** el programador que la utiliza (mediante la herencia) se ve obligado a proveer los métodos faltantes. Por lo tanto, ésta es una forma mediante la cual el diseñador de una clase puede fomentar un diseño en particular.

Se utiliza el término *abstracto* ya que, a medida que buscamos cada vez más alto en la jerarquía de clases, éstas se vuelven cada vez más generales o abstractas. En el ejemplo anterior, la clase **Forma** es más abstracta y menos concreta que la clase **Círculo**. La superclase abstrae las características (como la posición y el tamaño en este ejemplo) que son comunes entre sus subclases. Es común en

los programas OO extensos descubrir que los primeros niveles de las jerarquías de clases constan de métodos abstractos. De manera similar en la biología, la abstracción se utiliza en clases como la de los mamíferos, que no existen (por sí solas) pero sirven como superclases abstractas para un diverso conjunto de subclases. Por ejemplo, nunca hemos visto un objeto mamífero, pero hemos visto una vaca, que es una instancia de una subclase de mamífero.

PRÁCTICA DE AUTOEVALUACIÓN

- 10.4 Escriba una clase llamada **Cuadrado** que utilice la clase abstracta **Forma** antes descrita.

Principios de programación

Escribir programas como una colección de clases implica que los programas son modulares. Otro de los beneficios es que las partes de un programa se pueden reutilizar en otros programas. La herencia es otra forma en la que la POO provee el potencial de la reutilización. Algunas veces los programadores se ven tentados a reinventar la rueda: desean escribir nuevo software cuando simplemente podrían utilizar el software existente. Una de las razones para escribir nuevo software es que es divertido. El problema es que cada vez se está volviendo más complejo, por lo que simplemente no hay suficiente tiempo para escribirlo desde cero. Imagine tener que escribir el software para crear los componentes de GUI que proporcionan las bibliotecas de Java. Imagine tener que escribir una función matemática como `sqrt` cada vez que la necesite. Simplemente se llevaría demasiado tiempo. Por ende, una buena razón para reutilizar software es ahorrar tiempo. No sólo nos ahorraremos el tiempo para escribir el software sino también el tiempo para probarlo exhaustivamente, lo cual puede requerir aún más tiempo que el mero proceso de escribir el software. De aquí que la reutilización de clases tenga sentido.

Una razón por la que algunas veces los programadores no reutilizan el software es debido a que el software existente no hace exactamente lo que necesitan. Tal vez haga el 90% de lo que quieren, pero faltan ciertas características imprescindibles o algunas otras hacen las cosas de manera distinta. Una metodología sería modificar el software existente para que coincida con las nuevas necesidades. Sin embargo, ésta es una peligrosa estrategia ya que el proceso de modificar software es como estar en un campo de minas. El software es más quebradizo que suave; al tratar de modificarlo, se rompe. Al modificar el software es muy fácil introducir errores nuevos y sutiles, los cuales requieren de un extenso proceso de depuración y corrección. Ésta es una experiencia común, tanto que los programadores se muestran muy reacios a modificar el software.

Aquí es donde la POO entra en acción. En un programa OO podemos heredar el comportamiento de aquellas partes de cierto software que necesitamos, redefinir los (pocos) métodos de los cuales requerimos un comportamiento distinto, y agregar nuevos métodos que realicen cosas adicionales. A menudo podemos heredar la mayor parte de una clase y realizar sólo unos cuantos cambios que sean necesarios mediante la herencia. Sólo tenemos que probar las piezas nuevas, con la seguridad de que el resto ya ha sido probado. Entonces el problema de la reutilización queda resuelto. Podemos utilizar el software existente en forma segura. Mientras tanto, la clase original permanece intacta, confiable y utilizable.

La herencia es como ir a comprar un automóvil. Usted puede ver el automóvil que quiere, y es casi perfecto pero le gustaría realizar unas pequeñas modificaciones, como un color dis-





Principios de programación (continúa)

tinto o incluir la navegación por satélite. Con la herencia puede heredar el automóvil estándar y modificar las partes según lo requiera.

La POO implica basarse en el trabajo de otros. El programador OO procede de esta forma:

1. Aclara los requerimientos del programa.
2. Explora la biblioteca en busca de clases que realicen las funciones requeridas y las utiliza para obtener los resultados deseados.
3. Revisa las clases de otros programas que ha escrito y las utiliza de la manera apropiada.
4. Extiende las clases de biblioteca o sus propias clases mediante la herencia cuando es útil.
5. Escribe sus propias clases nuevas.

Esto explica por qué los programas OO son con frecuencia muy cortos: simplemente utilizan las clases de biblioteca o crean nuevas clases que heredan de las clases de biblioteca. Esta metodología requiere una inversión en tiempo: el programador necesita un conocimiento profundo de las bibliotecas. Esta idea de reutilizar software OO es tan poderosa que algunas personas piensan en la POO sólo de esta forma. Desde este punto de vista, la POO es el proceso de extender las clases de biblioteca de manera que cumplan los requerimientos de una aplicación específica.

Casi todos los programas de este libro utilizan la herencia. Cada programa empieza con una línea muy similar a la siguiente:

```
public class Bóveda extends JFrame
```

Esta línea indica que la clase **Bóveda** hereda características de la clase de biblioteca **JFrame**. Las herramientas de **JFrame** incluyen métodos para crear una ventana de GUI con los iconos usuales para cambiar el tamaño de la ventana y cerrarla. Extender la clase **JFrame** es la principal forma en la que los programas de este libro extienden las clases de biblioteca.

Tenga cuidado: algunas veces la herencia no es la técnica apropiada. La composición (utilizar las clases existentes sin modificarlas) es una alternativa que puede ser más conveniente en ocasiones. En el capítulo 18, sobre diseño, veremos esta cuestión con más detalle.

Errores comunes de programación

Los programadores principiantes utilizan la herencia de una clase de biblioteca, la clase **JFrame**, desde su primer programa. Pero aprender a utilizar la herencia dentro de nuestras propias clases es algo que lleva tiempo y experiencia. Por lo general, sólo vale la pena usarla en programas extensos. No se preocupe si no utiliza la herencia durante sus primeros programas.

Es común confundir la sobrecarga con la redefinición:

- **Sobrecargar** significa escribir dos o más métodos en la misma clase con el mismo nombre (pero una lista distinta de parámetros).
- **Redefinir (sobreescribir)** significa escribir un método en una subclase para utilizarlo en vez del método de la superclase (o de una de las superclases por encima de ella en el árbol de herencia).

Nuevos elementos del lenguaje

- **extends**: indica que esta clase hereda de otra clase.
- **protected**: la descripción de una variable o un método que se pueden utilizar dentro de la clase o de cualquier subclase (pero no desde ningún otro lado).
- **abstract**: la descripción de una clase abstracta que no se puede crear, es decir, que no es instanciable, sino que se proporciona sólo para utilizarse en la herencia.
- **abstract**: la descripción de un método que se proporciona sólo como encabezado y debe ser proporcionado por una subclase.
- **super**: el nombre de la superclase de una clase, la clase de la que hereda.
- **final**: describe a un método o una variable que no se puede redefinir.

Resumen

Extender (heredar) las herramientas de una clase es una buena forma de utilizar las partes existentes de los programas (clases).

Una subclase hereda las herramientas de su superclase inmediata y de todas las superclases por encima de ella en el árbol de herencia.

Una clase sólo tiene una superclase inmediata (sólo puede heredar de una clase). A esto se le conoce como *herencia simple* en la jerga de la POO.

Una clase puede extender las herramientas de una clase existente si proporciona uno o más de los siguientes elementos:

- Métodos adicionales.
- Variables adicionales.
- Métodos que redefinen a (actúan en vez de) los métodos de la superclase.

Una variable o un método se pueden describir con uno de los siguientes tres tipos de acceso:

- **public**: se puede utilizar desde cualquier clase.
- **private**: se puede utilizar sólo dentro de esta clase.
- **protected**: se puede utilizar sólo dentro de esta clase y de cualquier subclase.

Un diagrama de clases es un árbol que muestra las relaciones de herencia.

Para hacer referencia al nombre de la superclase de una clase se utiliza la palabra clave **super**.

Una clase abstracta se describe como **abstract**. No se puede instanciar para producir un objeto, ya que está incompleta. Dicha clase provee variables y métodos útiles que las subclases pueden heredar.

Ejercicios

- 10.1 Nave espacial** Escriba una clase llamada **NaveEspacial** que describa a una nave espacial. Se debe comportar de la misma forma que un objeto **Esfera**, sólo que se puede mover hacia arriba y hacia abajo. Recurra a la herencia para heredar de la clase **Esfera** que se muestra en el texto.

Dibuje un diagrama de clases para mostrar cómo se relacionan las distintas clases.

- 10.2 Fútbol** Escriba una clase llamada **Fútbol** que restrinja los movimientos de una pelota, de manera que la coordenada x deba ser mayor o igual a 0 y menor o igual a 200, lo que corresponde a la longitud de una cancha de fútbol. Recurra a la herencia para heredar de la clase **Pelota**.

- 10.3 El banco** Una clase describe cuentas bancarias y provee los métodos **abonarCuenta**, **cargarCuenta**, **calcularInterés** y **getSaldoActual**. Una nueva cuenta se crea con un nombre y un saldo inicial.

Hay dos tipos de cuentas: una cuenta regular y una cuenta dorada. La cuenta dorada produce un interés del 5%, mientras que la cuenta regular produce un interés del 6% menos un cargo fijo de \$100. Cada vez que se hace un retiro, se revisa una cuenta regular para ver si está sobregirada. El cliente con una cuenta dorada puede sobregirarse en forma indefinida.

Escriba clases que describan los dos tipos de cuentas y utilice una clase abstracta para describir las características comunes (para simplificar, asuma que las cantidades de dinero se guardan como valores **int**).

- 10.4 Formas** Escriba una clase abstracta llamada **Forma** para describir objetos gráficos bidimensionales (cuadrado, círculo, rectángulo, triángulo, etc.) que tengan las siguientes características. Todos los objetos usan variables **int** que especifican las coordenadas x y y de la esquina superior izquierda de un rectángulo circundante, además de variables **int** que describen la altura y la anchura del rectángulo. Todos los objetos comparten los mismos métodos **setX** y **setY** para establecer los valores de estas coordenadas. Todos los objetos comparten los métodos **setAnchura** y **setAltura** para establecer los valores de la anchura y la altura del objeto. Todos los objetos tienen un método llamado **getÁrea** que devuelve el área del objeto, junto con un método **mostrar** para mostrarlo en pantalla, pero estos métodos son distintos dependiendo del objeto específico.

Escriba una clase llamada **Rectángulo** que herede de la clase **Forma**.

Respuestas a las prácticas de autoevaluación

- 10.1 public class Pelota extends Esfera {
 public void moverIzquierda(int cantidad) {
 x = x - cantidad;
 }

 public void moverDerecha(int cantidad) {
 x = x + cantidad;
 }
 }

10.2 public class EsferaColoreada extends Esfera {
 private Color color;

 public EsferaColoreada(Color colorInicial) {
 color = colorInicial;
 }
 }

10.3 El compilador encontrará una falla en la subclase. No hay una llamada explícita a un constructor de la superclase, por lo que Java tratará de llamar a un constructor sin parámetros de la superclase y no se ha escrito dicho método.

10.4 import java.awt.*;

 public class Cuadrado extends Forma {
 public void mostrar(Graphics papel) {
 papel.drawRect(x, y, tamaño, tamaño);
 }
 }