CAPÍTULO

4 Herencia

Tal y como se ha comentado en el Capítulo 3, un objetivo importante de la programación orientada a objetos es la reutilización de código. Igual que los ingenieros emplean repetidamente en sus diseños las mismas componentes, los programadores deberían ser capaces de reutilizar objetos, en lugar de implementarlos repetidamente. En un lenguaje de programación orientado a objetos, el mecanismo básico de reutilización de código es la *herencia*. La herencia permite extender la funcionalidad de un objeto. En otras palabras, podemos definir nuevos tipos restringiendo (o aumentando) propiedades del tipo original, generándose así una jerarquía de clases. La herencia es también el mecanismo que Java emplea para implementar métodos y clases genéricos.

En este capítulo veremos:

- Los principios generales de la herencia, incluyendo el *polimorfismo*.
- Cómo se implementa la herencia en Java.
- Cómo puede derivarse una colección de clases a partir de una única clase abstracta.
- La *interfaz*, que es un tipo especial de clase.
- Cómo implementa Java la programación genérica usando herencia.

4.1 ¿Qué es la herencia?

La *herencia* es el principio básico de la programación orientada a objetos empleado para reutilizar código entre clases relacionadas. La herencia modela las *relaciones ES-UN(A)*. En una relación ES-UN(A), decimos que una clase derivada *es una* variación de la clase base. Por ejemplo, un Círculo ES-UN(A) Figura y un Coche ES-UN(A) Vehículo. Sin embargo, no se tiene que una Elipse ES-UN(A) Círculo. Las relaciones de herencia forman *jerarquías*. Por ejemplo, podemos extender Coche formando otras clases, ya que CocheExtranjero ES-UN(A) Coche (y paga impuestos) y CocheNacional ES-UN(A) Coche (y no paga impuestos).

En una *relación ES-UN(A)* decimos que la clase derivada *es una* variación de la clase base.

En una *relación CONTIENE* decimos que la clase derivada *contiene* una instancia de la clase base. Las relaciones de este tipo se modelan mediante *composición*.

Otro tipo de relaciones son las relaciones *CONTIENE*. Esta clase de relaciones no posee las propiedades asociadas a la jerarquía de herencia. Un ejemplo de relación CONTIENE es que un Coche CONTIENE Volante. Las relaciones CONTIENE no deben ser modeladas a través de la herencia. Por el contrario, deben implementarse mediante la técnica de la *composición*, en la que las componentes son sencillamente atributos de datos privados.

El lenguaje Java emplea extensamente la herencia al implementar sus librerías de clases. Dos ejemplos son las *excepciones* y las *componentes* en el *Abstract Window Toolkit*:

- *Excepciones*. Java define la clase `Exception`. Como ya hemos visto, existe una gran variedad de excepciones incluyendo `NullPointerException` y `ArrayOutOfBoundsException`. Cada una es una clase distinta, pero sobre todas ellas pueden aplicarse los métodos `toString` y `printStackTrace` para ayudarnos en la depuración.
- *Componentes*. El *Abstract Window Toolkit* (descrito con más detalle en el Apéndice D) define un objeto conocido como `Component`. Existen muchos tipos de `Componentes`, incluyendo `Button`, `Canvas`, `Checkbox`, `List` y `TextField`. Cualquiera de ellas puede añadirse a un `Frame`, `Panel` o `Window` (que a su vez son `Componentes`).

En ambos casos, la herencia modela una relación ES-UN(A). Un `Button` ES-UN(A) `Component`. Una `NullPointerException` ES-UN(A) `Exception`. Debido a las características de la relación ES-UN(A), la propiedad fundamental de la herencia garantiza que cualquier método ejecutable por `Exception` puede ser ejecutado también por `NullPointerException`, y más aún, un objeto `NullPointerException` puede estar referenciado por una variable `Exception`. (Observe que lo contrario no es cierto.) Entonces, como `printStackTrace` es un método disponible en la clase `Exception`, podemos escribir

```
catch( Exception e ) { e.printStackTrace( ); }
```

Si `e` referencia a un objeto `NullPointerException`, `e.printStackTrace` tiene sentido. Dependiendo de las circunstancias de la jerarquía de clases que se tenga, el método `printStackTrace` puede permanecer igual o *especializarse* en cada clase. Cuando un método permanece invariante a lo largo de la jerarquía, en el sentido de tener la misma funcionalidad para todas las clases en la jerarquía, evitamos tener que rescribir una implementación del método para cada clase.

Una *variable polimórfica* puede referenciar objetos de distintos tipos. Cuando se aplican operaciones sobre las variables polimórficas, se selecciona automáticamente la más adecuada en ese momento.

La llamada a `printStackTrace` muestra también un importante principio de la programación orientada a objetos conocido como *polimorfismo*. Una variable referencia polimórfica puede referenciar objetos de distintos tipos. Cuando una operación ha de aplicarse a una referencia tal, se selecciona automáticamente la más adecuada al objeto referenciado en ese momento. En Java todos los tipos referencia son polimórficos. En el caso de una referencia de tipo `Exception` se toma una decisión en tiempo de ejecución: el método `printStackTrace` del objeto que `e` referencia en tiempo de ejecución es el empleado. Esto es conocido como *tipado dinámico*.

En la *herencia*, se tiene una *clase base* de la que derivamos el resto de clases. La clase base es la clase sobre la que se basa la herencia. Una *clase derivada* hereda todas las propiedades de la clase base, de modo que todos los métodos públicos

disponibles en la clase base se convierten en métodos públicos, con implementaciones idénticas en la clase derivada. Además pueden añadirse atributos y métodos adicionales y cambiar el significado de los métodos heredados. Cada clase derivada es una clase completamente nueva. Por otra parte, los cambios realizados en las clases derivadas no afectan en absoluto a la clase base. De este modo, el diseño de clases derivadas no afecta a la clase base, lo que simplifica notablemente el mantenimiento del software.

La *herencia* permite derivar clases a partir de una *clase base* sin modificar la implementación de esta última.

Una clase derivada es de tipo compatible con la clase base, de modo que una variable referencia de la clase base puede referenciar a un objeto de la clase derivada, pero no al revés. Las clases hermanas (es decir, las clases derivadas de una clase común) no son de tipos compatibles.

Cada *clase derivada* es una clase completamente nueva, compatible con la clase base de la que se deriva.

Como hemos mencionado anteriormente, el uso de la herencia genera una jerarquía de clases. La Figura 4.1 muestra una pequeña parte de la jerarquía `Exception`. Nótese que `NullPointerException` se deriva indirectamente de `Exception`. Este hecho es transparente al usuario de las clases ya que las relaciones ES-UN(A) son transitivas. En otras palabras, si *X* ES-UN(A) *Y* e *Y* ES-UN(A) *Z*, entonces *X* ES-UN(A) *Z*. La jerarquía `Exception` muestra las decisiones de diseño usuales en las que los rasgos comunes se reúnen en las clases base y la especialización se hace en las clases derivadas. En esta jerarquía se dice que la clase derivada es una *subclase* de la clase base y que la clase base es una *superclase* de la clase derivada. Estas relaciones son transitivas. Por otra parte, el operador `instanceof` puede emplearse con las subclases. Así, si `obj` es de tipo `X` (y no `null`) entonces `obj instanceof Z` es `true`.

Si *X* ES-UN(A) *Y* entonces *X* es una *subclase* de *Y* e *Y* es una *superclase* de *X*. Ambas relaciones son transitivas.

En las siguientes secciones responderemos a las siguientes cuestiones:

- ¿Cuál es la sintaxis adecuada para derivar una clase de otra clase base ya existente?
- ¿Cómo afecta esto a lo público y lo privado?
- ¿Cómo se especifica que un método no se redefine a lo largo de la jerarquía de clases?
- ¿Cómo se especializa un método?
- ¿Cómo podemos eliminar las diferencias generando una clase abstracta para crear después una jerarquía?
- ¿Se puede derivar una nueva clase a partir de varias clases (*herencia múltiple*)?
- ¿Cómo se emplea la herencia para implementar la genericidad?

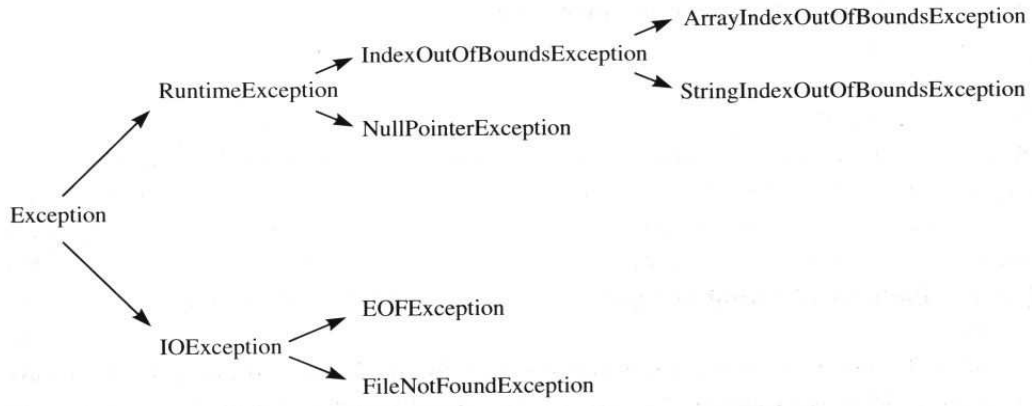

Figura 4.1 Parte de la jerarquía `Exception`.

Algunas de estas cuestiones se ilustran en la implementación de la clase `Figura` (en las Secciones 4.2.5 y 4.3) de la que se derivan `Circulo`, `Cuadrado` y `Rectangulo`. En este ejemplo puede observarse cómo Java implementa el polimorfismo en tiempo de ejecución, y comprobar cómo puede emplearse la herencia para implementar métodos genéricos.

4.2 Sintaxis básica de Java

La *cláusula* `extends` se emplea para indicar que una clase se deriva a partir de otra.

Recuerde que una clase derivada hereda todas las propiedades de cada una de sus clases base. Después podemos añadir nuevos atributos, sobrescribir métodos y añadir nuevos métodos. Cada clase derivada es una clase completamente nueva. En la Figura 4.2 se muestra un esqueleto típico de herencia, en el que se usa la *cláusula* `extends`. Una cláusula `extends` indica que una clase se deriva de otra. Una clase derivada *extiende* una clase base. Presentamos a continuación una breve descripción genérica de una clase derivada:

Una clase derivada hereda todos los atributos de la clase base y puede añadir atributos adicionales. La clase derivada hereda todos los métodos de la clase base. Puede quedarse con dichos métodos sin más o redefinirlos. También puede incorporar nuevos métodos.

- Generalmente todos los datos son privados, así que al añadir nuevos atributos a la clase derivada debemos especificarlos en la sección privada.
- Cualquier método de una clase base que no se especifique en la clase derivada es heredado sin modificación alguna, a excepción del constructor. Este caso especial se detalla en la Sección 4.2.2.
- Cualquier método de una clase base que se defina de nuevo en la sección pública de la clase derivada se sobrescribe. La nueva definición se aplicará a los objetos de la clase derivada.
- Los métodos públicos de la clase base no deben redefinirse en la sección privada de la clase derivada.
- En la clase derivada puede añadirse métodos adicionales.

```
1  public class Derivada extends Base
2  {
3      // Cualquier componente no listada se hereda sin cambios
4      // excepto el constructor
5
6          // componentes públicos
7      // Constructor(es) si el constructor por defecto no es válido
8      // Métodos de la clase base redefinidos
9      // Métodos públicos adicionales
10
11         // componentes privados
12     // Atributos adicionales (generalmente privados)
13     // Métodos privados adicionales
14 }
```

Figura 4.2 Esquema general de la herencia.

4.2.1 Reglas de visibilidad

Ya sabemos que cualquier componente privada es accesible sólo para los métodos de su clase. De este modo, las componentes privadas de la clase base no son accesibles desde la clase derivada.

Es posible que en ocasiones necesitemos que la clase derivada tenga acceso a algunas componentes de su clase base. Existen dos formas de lograrlo. La primera consiste en emplear el acceso público o el acceso amistoso. Sin embargo, el acceso público permite acceder a otras clases además de a la clase derivada, y el acceso amistoso sólo funciona si ambas clases están en el mismo paquete.

Si deseamos permitir el acceso sólo a las clases derivadas, podemos declarar componentes protegidas. Una *componente protegida* de una clase es privada para cualquier clase salvo para las clases derivadas (y las clases que se encuentren en el mismo paquete). El declarar componentes como públicas o protegidas viola el espíritu de la encapsulación y el ocultamiento de información, por lo que sólo se debería admitir como licencia. Una mejor alternativa es escribir métodos de modificación o de acceso para aprovechar el acceso amistoso. Sin embargo, si una declaración protegida permite evitar un código complicado, es razonable emplearla. En este texto, utilizaremos en ocasiones atributos protegidos por esta razón. El empleo de métodos protegidos es también una práctica habitual en el texto. Esto permite que una clase derivada herede un método interno sin convertirlo en accesible fuera de la jerarquía de paquetes.

Una *componente protegida de una clase* es privada para cualquier clase excepto para las clases derivadas (y aquéllas que se encuentren en el mismo paquete).

4.2.2 El constructor y `super`

Cada clase derivada debe definir sus propios constructores. Si no se implementa ninguno entonces se genera un constructor sin parámetros por defecto. Este constructor invocará al constructor sin parámetros de la clase base para inicializar la parte heredada, y se empleará la inicialización por defecto sobre los nuevos atributos (o sea, utilizará 0 para los tipos primitivos y `null` para los tipos referencia).

En la práctica es habitual construir una clase derivada construyendo primero la parte heredada. De hecho, esto se hace así siempre por defecto, incluso cuando se tiene un constructor para la clase derivada. Esto es lógico si se tiene en cuenta que la encapsulación considera la porción heredada como una sola entidad, y el constructor de la clase base indica como inicializar dicha entidad.

Los constructores de la clase base pueden ser llamados explícitamente empleando el método `super`. Así, el constructor por defecto de una clase derivada es en realidad

Si no se implementa ningún constructor, se genera por defecto un constructor sin parámetros. Dicho constructor invoca al constructor sin parámetros de la clase base para inicializar la porción heredada, mientras que para los atributos adicionales se emplea la inicialización por defecto.

```
public Derivada( )
{
    super( );
}
```

La llamada al método `super` puede realizarse con los parámetros que se ajusten al constructor de la clase base. Por ejemplo, la Figura 4.3 muestra la clase `DesbordamientoInferior` empleada en la implementación de las estructuras de datos. `DesbordamientoInferior` se lanza cuando se intenta obtener información de una estructura de datos vacía. Los objetos de tipo `DesbordamientoInferior` se crean especificando un `String`. `DesbordamientoInferior` no incorpora nuevos atributos, por lo que la construcción de objetos de dicha subclase se reduce a construir el fragmento heredado, empleando el constructor de `Exception`.

`super` se emplea para llamar al constructor de la clase base.

```
1 package Excepciones;
2
3 public class DesbordamientoInferior extends Exception
4 {
5     public DesbordamientoInferior( String lanzador )
6     {
7         super( lanzador );
8     }
9 }
```

Figura 4.3 Constructor para una nueva excepción `DesbordamientoInferior` que emplea `super`.

El método `super` sólo puede emplearse como primera línea de un constructor. Si éste no aparece se genera de forma automática una llamada sin parámetros a `super`.

4.2.3 Métodos y clases finales

Un *método final* permanece invariable a lo largo de la jerarquía de clases y no debe ser redefinido.

Como se ha descrito anteriormente, la clase derivada o bien sobrescribe o bien deja intactos los métodos de la clase base. En muchos casos, está claro que un determinado método de la clase base debe permanecer invariante a lo largo de la jerarquía de clases, esto es, que ninguna clase derivada debe redefinirlo. En este caso podemos declarar que el método es *`final`*, de modo que no podrá sobrescribirse.

La declaración de un método invariante como `final` no es sólo una buena costumbre de programación, sino que puede conducirnos a código más eficiente. Es una buena costumbre de programación, porque además de declarar nuestras intenciones al lector del programa y de la documentación, se evita la redefinición accidental de un método que no debe ser rescrito.

Para comprobar por qué el uso de `final` puede generar código más eficiente, suponga que una clase base `Base` declara un método final `f` y que `Derivada` extiende a la clase `Base`.

Considere la rutina

```
void hazlo( Base obj )
{
    obj.f( );
}
```

El tipado estático se emplea cuando un método permanece invariante a lo largo de la jerarquía de clases.

Como `f` es un método final, no importa si `obj` referencia a un objeto de `Base` o `Derivada`; la definición de `f` es invariante, de modo que sabremos lo que hará `f`. Como consecuencia, la llamada al método se resuelve en tiempo de compilación en lugar de en tiempo de ejecución. Esto se conoce por *tipado estático*. El programa se ejecutará más rápido, ya que los tipos se determinan durante la compilación y no durante la ejecución. En qué grado esto es importante depende de cuántas veces evitemos tomar decisiones durante la ejecución del programa.

Un corolario de esta observación es que si `f` es un método trivial, tal como un mero acceso a un atributo, y se declara como `final`, el compilador puede reemplazar la llamada a `f` por su definición. Así, la llamada al método será sustituida por una sola línea que realiza el acceso al atributo, ganando tiempo. Si `f` no se declara como `final` esto es imposible, ya que `obj` podría referenciar a un objeto de la clase derivada, para el que la definición de `f` sería diferente. Los métodos estáticos no necesitan de objeto controlador y sus llamadas se resuelven en tiempo de compilación mediante el tipado estático.

Los métodos estáticos no tienen un objeto controlador y las llamadas a los mismos se resuelven en tiempo de compilación empleando el tipado estático.

Las *clases finales* son similares a los métodos finales. Una clase final no puede ser extendida. Como consecuencia, todos sus métodos son automáticamente métodos finales. Por ejemplo, la clase `Integer` es una clase final. Nótese que el hecho de que una clase tenga métodos finales no implica que la clase sea también final. Las clases finales se conocen también por *clases hoja* ya que en la jerarquía de la herencia, que se asemeja a un árbol, las clases finales están en los bordes, como las hojas.

Una *clase final* no debe extenderse. Una *clase hoja* es una clase final.

4.2.4 Sobrescribiendo un método

Los métodos de la clase base se sobrescriben en la clase derivada con sólo definir en la clase derivada un método con la misma signatura. El método de la clase derivada debe tener el mismo tipo del retorno y no puede añadir excepciones en la lista `throws`.

El método de la clase derivada debe tener el mismo tipo de resultado y la misma signatura y no debe añadir excepciones a la lista `throws`.

En algunas ocasiones los métodos de la clase derivada necesitan invocar a los métodos de la clase base. Esto se conoce con el nombre de *sobrescritura parcial*. Esto es, deseamos hacer lo mismo que la clase base y un poco más, en lugar de hacerlo todo de manera completamente diferente. Las llamadas a un método de la clase base se hacen a través de `super`. Aquí se muestra un ejemplo:

La *sobrescritura parcial* incluye una llamada al método de la clase base usando `super`.

```
public class TrabajadorAdicto extends Trabajador
{
    public trabaja( )
    {
        super.trabaja( ); // Trabaja como un Trabajador
        bebeCafe( );      // Se toma un descanso
        super.trabaja( ); // Trabaja como un Trabajador un poco más
    }
}
```

4.2.5 Métodos y clases abstractos

Ya hemos visto que algunos métodos permanecen invariantes a lo largo de la jerarquía de clases (éstos son los métodos finales), y que otros pueden ver modificado su significado en la jerarquía. Una tercera posibilidad es que el método tenga sentido para las clases derivadas y que deba implementarse en ellas, mientras que su implementación no tiene sentido en la clase base. En este caso podemos declarar el método en la clase base como *abstracto* mediante la palabra reservada `abstract`.

Un *método abstracto* no tiene ninguna definición concreta en la clase abstracta y debe definirse en la clase derivada.

Un *método abstracto* es un método que declara una funcionalidad que debe ser implementada en todas las clases derivadas de la clase en que se vaya a utilizar. En otras palabras, a través de un método tal indicamos lo que deberían poder hacer los objetos de dichas subclases. No se implementa en la clase base, sino que cada subclase debe tener su propia implementación.

Una clase que tiene al menos un método abstracto es una *clase abstracta*. En Java se exige que todas las clases abstractas sean declaradas como tales. Cuando en una clase derivada no se (re)define un método abstracto mediante una implementación, el método continúa siéndolo en la clase derivada. Como resultado, si en una clase no definida como abstracta no se implementa un método abstracto, el compilador detectará esa inconsistencia y devolverá un error.

Un ejemplo viene dado por la clase abstracta `Figura`, que se empleará en un ejemplo mayor de este capítulo, un poco más tarde. Las figuras concretas, como `Circulo` o `Rectangulo`, se derivan de `Figura`. `Cuadrado` puede derivarse como un `Rectangulo` especial. La Figura 4.4 ilustra la jerarquía resultante.

La clase `Figura` puede contener atributos comunes a todas las clases. En un ejemplo más concreto podrían incluirse como tales las coordenadas de los extremos de los objetos. La citada clase da la definición de algunos métodos, como `posicionDe`, que son independientes del tipo del objeto referenciado en cada momento; `posicionDe` será entonces un método final. También se declaran métodos que se aplican a cada tipo concreto de objetos. Algunos de estos métodos podrían ser implementados genéricamente en la clase abstracta `Figura`. Por lo que, por ejemplo, es imposible indicar cómo calcular el área de un objeto abstracto; el método `area` debería ser declarado abstracto mediante `abstract`.

Una clase con, al menos, un método abstracto debe ser una *clase abstracta*.

Como ya hemos mencionado anteriormente, la existencia de un método abstracto convierte a la clase en abstracta, no permitiéndose la creación de objetos de dicha clase. De este modo, no podemos construir un objeto de la clase `Figura`; sólo podemos crear objetos de las clases derivadas. Sin embargo, como de costumbre, una `Figura` puede referenciar cualquier objeto concreto de las clases derivadas, como `Circulo` o `Rectangulo`. Así,

```
Figura a, b;
a = new Circulo( 3.0 );          // Correcto
b = new Figura ( "Circulo" );    // Incorrecto
```

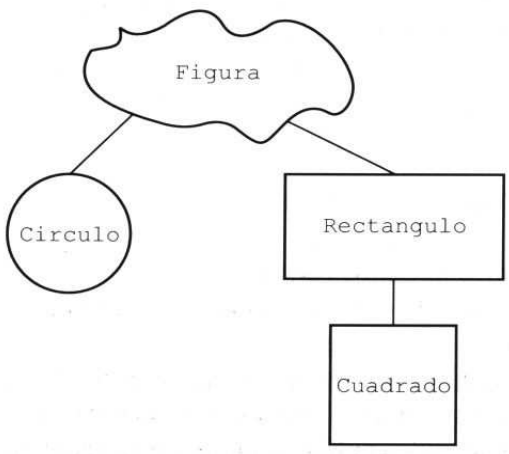

Figura 4.4 La jerarquía de las figuras.

```
1  // Clase base abstracta para las figuras
2  //
3  // CONSTRUCCIÓN: no se permite. Figura es un constructor abstracto
4  //     con un solo parámetro invocado por las clases derivadas
5  //
6  // ********************OPERACIONES PÚBLICAS********************
7  // double area( )            --> Devuelve el área (abstracto)
8  // boolean menorQue( lder ) --> Compara el área de dos objetos
9  // String toString( )        --> Método toString usual
10
11 public abstract class Figura
12 {
13     abstract public double area( );
14
15     public Figura( String nombreFigura )
16     { nombre = nombreFigura; }
17
18     final public boolean menorQue( Figura lder )
19     { return area( ) < lder.area( ); }
20
21     final public String toString( )
22     { return nombre + " con área " + area( ); }
23
24     private String nombre;
25 }
```

Figura 4.5 La clase abstracta `Figura`.

La Figura 4.5 muestra la clase abstracta `Figura`. En la línea 24 se declara un `String` que almacena el tipo de la figura. Esto se emplea únicamente en las clases derivadas. Este atributo es privado, así que las clases derivadas no tienen acceso directo a él. El resto de la clase especifica una colección de métodos.

El constructor nunca se invoca directamente, ya que `Figura` es una clase abstracta. Sin embargo, necesitamos un constructor, al que la clase derivada invocará cuando necesite inicializar los componentes privados. El constructor de `Figura` inicializa el atributo interno `nombre`.

Los objetos de una clase abstracta no pueden construirse directamente. Sin embargo, se sigue definiendo un constructor que es empleado por las clases derivadas.

La línea 13 declara el método abstracto `area`. En tiempo de ejecución se decidirá el método `area` adecuado de la correspondiente clase derivada. `area` es un método abstracto, ya que no tiene ningún significado por defecto que pueda ser especificado para aplicarse sobre una clase heredada que opte por no implementarlo.

El método de comparación de figuras implementado en las líneas 18 y 19, no es abstracto ya que puede ser aplicado en todas las clases derivadas. De hecho, su definición es invariante a lo largo de la jerarquía de figuras, así que lo hemos convertido en un método final.

El método `toString`, mostrado en las líneas 21 y 22, imprime el nombre de una figura y su área. Al igual que el método de comparación, permanece invariante a lo largo de la jerarquía completa, y se declara como final.

Antes de continuar, vamos a resumir los distintos tipos de métodos existentes:

1. *Métodos finales*. La sobrecarga se resuelve en tiempo de compilación. Empleamos un método final sólo cuando dicho método es invariante a lo largo de la jerarquía de herencia (esto es, cuando el método no se redefine nunca).

2. *Métodos abstractos*. La sobrecarga se resuelve durante la ejecución del programa. La clase base no implementa estos métodos y es abstracta a su vez. Es necesario que las clases derivadas implementen dichos métodos o sean también abstractas.
3. *Métodos estáticos*. La sobrecarga se resuelve durante la compilación ya que no existe ningún objeto controlador.
4. *Otros métodos*. La sobrecarga se resuelve en tiempo de ejecución. La clase base tiene una implementación por defecto que puede ser, o bien sobrescrita en las clases derivadas, o permanecer sin cambios en ellas.

4.3 Ejemplo: extensión de la clase `Figura`

En esta sección implementamos las clases derivadas de `Figura` y mostramos cómo se emplean de manera polimórfica. Para ello consideramos el siguiente problema:

ORDENANDO FIGURAS
Léanse los datos de N figuras (círculos, cuadrados o rectángulos) y muéstrense ordenadas por área.

La implementación de las clases derivadas, mostrada en la Figura 4.6, es muy sencilla y no incluye nada que no haya sido comentado anteriormente. El único punto novedoso es que `Cuadrado` se define a partir de `Rectangulo`, que se deriva a su vez de `Figura`. Esta derivación se realiza exactamente igual que cualquier otra. En la implementación de estas clases debemos hacer lo siguiente:

1. Definir un nuevo constructor.
2. Estudiar cada método que no sea ni abstracto ni final para decidir si se acepta su definición por defecto. De no ser así, debemos escribir para él una nueva definición.
3. Escribir una definición para cada método abstracto.
4. Si es necesario, definir métodos adicionales.

Para cada clase hemos definido un sencillo constructor en el que se inicializan las dimensiones básicas (el radio para los círculos, y las longitudes de los lados para cuadrados y rectángulos). Primero se inicializa la parte heredada empleando `super`. En cada clase debe definirse un método `area` ya que en `Figura` aparece como método abstracto. Si el método `area` no es implementado en alguna de las clases derivadas se producirá un error durante la compilación. Esto es debido a que si falta la implementación de `area`, la clase derivada también debería declararse como abstracta. Nótese que `Cuadrado` hereda el método `area` de `Rectangulo`, por lo que no es necesario implementarlo de nuevo.

Ahora que hemos escrito las clases, ya podemos resolver el problema propuesto. Para ello empleamos un vector de `Figuras`. Es importante observar que dicho vector no almacenará directamente `Figuras` (lo que no es posible) sino que guardará referencias de tipo `Figura`. Estas referencias pueden apuntar a `Circulos`, `Cuadrados` y `Rectangulos`.

```
1  // Clases Circulo, Cuadrado y Rectangulo
2  //      basadas todas en Figura
3  //
4  // CONSTRUCCIÓN: con un radio (para el círculo), longitud del lado
5  //      (para el cuadrado), y largo y ancho (para el rectángulo).
6  // ********************OPERACIONES PÚBLICAS*********************
7  // double area( ) --> Implementa el método area abstracto en Figura
8
9  public class Circulo extends Figura
10 {
11     public Circulo( double r )
12     {
13         super( "Círculo" );
14         radio = r;
15     }
16
17     public double area( )
18     {
19         return PI * radio * radio;
20     }
21
22     private static final double PI = 3.14159265358979323;
23     private double radio;
24 }
25
26 public class Rectangulo extends Figura
27 {
28     public Rectangulo( double largo, double ancho )
29     {
30         super( "Rectangulo" );
31         base = largo;
32         alt = ancho;
33     }
34
35     public double area( )
36     {
37         return base * alt;
38     }
39
40     private double base;
41     private double alt;
42 }
43
44 public class Cuadrado extends Rectangulo
45 {
46     public Cuadrado( double lado )
47     {
48     super( lado, lado );
49     }
50 }
```

Figura 4.6 Clases `Circulo`, `Cuadrado` y `Rectangulo`, situadas en ficheros distintos.

La Figura 4.7 muestra este enfoque del problema. En primer lugar, se realiza la lectura de los objetos. En la línea 23, la llamada a `leerFigura` consiste en leer un carácter y las dimensiones de una figura, para crear después un objeto de tipo

```
1  import java.io.*;
2
3  class TestFigura
4  {
5      private static BufferedReader in;
6
7      private static Figura leer Figura( )
8        { /* Implementación en la Figura 4.8 */
9
10     public static void main( String [ ] args )
11     {
12         try
13         {
14                 // Lectura del número de figuras
15             System.out.println( "Introduzca # de figuras: ");
16             in = new BufferedReader( new
17                                 InputStreamReader( System.in ) );
18             int numFiguras = Integer.parseInt( in.readLine( ) );
19
20               // Lectura de las figuras
21             Figura [ ] vector = new Figura [ numFiguras ];
22             for( int i = 0; i < numFiguras; i++ )
23                vector[ i ] = leerFigura( );
24
25               // Ordenación y Salida
26             ordenacionPorInsercion( vector );
27             System.out.println( "Ordenados por area" );
28             for ( int i = 0; i < numFiguras; i++)
29                 System.out.println( vector[ i ] );
30         }
31         catch( Exception e )
32         { System.out.println( e ); }
33     }
34 }
```

Figura 4.7 Rutina `main` para leer figuras y mostrarlas en orden creciente de áreas.

`Figura`. La Figura 4.8 muestra una implementación directa de ello. Nótese que si se produce una entrada de datos incorrectos, se crea un círculo de radio cero y se devuelve una referencia apuntando hacia él. Una solución más elegante en este caso sería definir y lanzar una excepción. Haga esto como Ejercicio 4.14.

A continuación, cada objeto creado en `leerFigura` se referencia a través de una posición del vector. Después la llamada a `ordenacionPorInsercion` ordena las figuras. Finalmente, se muestra por pantalla el vector de `Figuras`, llamando implícitamente al método `toString`.

4.3.1 Disgresión: una introducción a la ordenación

La ordenación por inserción es un algoritmo de ordenación sencillo apropiado para entradas pequeñas.

La ordenación en el ejemplo anterior se ha implementado usando un algoritmo denominado *ordenación por inserción*. Esta técnica se considera una buena solución cuando son pocos los elementos a ordenar, ya que el algoritmo en cuestión es muy sencillo. Sin embargo, consume demasiado tiempo, y si queremos ordenar una gran cantidad de elementos, la ordenación por inserción es una pobre elección. En tal caso debemos emplear algoritmos mejores, tal y como se estudia en el Capítulo 8. La ordenación por inserción está implementada en la Figura 4.9.

```
1  // Crea la figura adecuada basándose en la entrada.
2  // Es parte de la clase TestFigura de la Figura 4.7.
3  // El usuario debe teclear 'c', 's' o 'r' para indicar la figura
4  // y después introducir las dimensiones.
5  // Ante cualquier error se devuelve un círculo de radio cero.
6
7     private static Figura leerFigura( )
8     {
9         double rad;
10        double largo;
11        double ancho;
12        String unaLinea;
13
14        try
15        {
16           System.out.println( "Introduzca el tipo de figura:" );
17           do
18           {
19               unaLinea = in.readLine( );
20           } while( unaLinea.length( ) == 0 );
21
22           switch( unaLinea.charAt( 0 ) )
23           {
24             case 'c':
25               System.out.println( "Introduzca el radio: " );
26               rad = Double.valueOf( in.readLine( .doubleValue( );
27               return new Circulo( rad );
28
29             case 's':
30               System.out.println( "Introduzca el lado: " );
31               largo=Double.valueOf(in.readLine()).doubleValue();
32               return new Cuadrado( largo );
33
34             case 'r':
35               System.out.println( "Introduzca el largo y el ancho "
36                                   + "en líneas separadas: " );
37               largo=Double.valueOf(in.readLine()).doubleValue();
38               ancho=Double.valueOf(in.readLine()).doubleValue();
39               return new Rectangulo( largo, ancho );
40
41             default:
42               System.err.println( "Me faltan c, r o s" );
43               return new Circulo( 0 );
44           }
45        }
46        catch( IOException e )
47        {
48           System.err.println( e );
49           return new Circulo( 0 );
50        }
51 }
```

Figura 4.8 Rutina para leer los datos de entrada y devolver una `Figura`.

La ordenación por inserción trabaja del siguiente modo. En el estado inicial se parte de que el primer elemento considerado aisladamente está ordenado. En el estado final que queremos obtener se tienen ordenados todos los elementos del vector. La Figura 4.10 muestra que la acción básica de la ordenación consiste en

```
1  // ordenacionPorInsercion: ordena el vector a.
2      private static void ordenacionPorInsercion( Figura [ ] a )
3      {
4          for( int p = 1; p < a.length; p++ )
5          {
6              Figura tmp = a[ p ];
7              int j = p;
8
9              for( ; j > 0 && tmp.menorQue( a[ j - 1 ] ); j-- )
10                 a[ j ] = a[ j - 1 ];
11             a[ j ] = tmp;
12         }
13     }
```

Figura 4.9 Ordenación por inserción.

Posición del vector	0	1	2	3	4	5
Estado inicial:	8	5	9	2	6	3
Tras ordenar a(0..1):	5	8	9	2	6	3
Tras ordenar a(0..2):	5	8	9	2	6	3
Tras ordenar a(0..3):	2	5	8	9	6	3
Tras ordenar a(0..4):	2	5	6	8	9	3
Tras ordenar a(0..5):	2	3	5	6	8	9

Figura 4.10 Acción básica de la ordenación por inserción (la parte sombreada está ya ordenada).

conseguir que los elementos entre las posiciones 0 y `p` (donde `p` toma valores entre 1 y $N-1$ y N es el número de elementos a ordenar) queden correctamente ordenados. En cada paso, `p` aumenta en una unidad, lo que es controlado en el bucle de la línea 4 en la Figura 4.9.

Cuando se ejecuta el cuerpo del bucle `for` en la línea 6, tenemos garantizado que los elementos del vector entre las posiciones 0 y `p` -1 están ya ordenados. La Figura 4.11 muestra más detalladamente lo que vamos haciendo, resaltando en cada caso la parte relevante del vector. En cada paso necesitamos añadir el elemento en negrita en la parte del vector previamente ordenada. Esto se logra fácilmente almacenando dicho elemento en una variable temporal y desplazando todos los elementos mayores que él una posición a la derecha. Después de esto, podemos copiar el contenido de la variable temporal en la posición anterior al elemento recolocado que esté más a la izquierda (indicado con un sombreado más claro en la línea siguiente). Se mantiene un contador `j`, que es la posición en la que debe colocarse la variable temporal. `j` decrece en una unidad cada vez que se desplaza un elemento del vector. Las líneas entre la 6 y la 11 implementan lo indicado.

Posición del vector	0	1	2	3	4	5
Estado inicial:	8	**5**				
Tras ordenar a(0..1):	5	8	**9**			
Tras ordenar a(0..2):	5	8	9	**2**		
Tras ordenar a(0..3):	2	5	8	9	**6**	
Tras ordenar a(0..4):	2	5	6	8	9	**3**
Tras ordenar a(0..5):	2	3	5	6	8	9

Figura 4.11 Detalle de la ordenación por inserción (el sombreado oscuro indica la parte ordenada; el sombreado claro indica dónde debe colocarse el nuevo elemento).

Deben comprobarse siempre los casos extremos.

Es importante comprobar que la ordenación por inserción se comporta correctamente en los dos casos extremos. En primer lugar, si en la Figura 4.11 el elemento en negrita es el mayor de los que se encuentran en el grupo, entonces se copia en la variable temporal e inmediatamente se vuelve a insertar en el vector, por lo que el algoritmo es correcto. Si en la misma figura, el elemento en negrita es el menor de todos los del grupo, entonces el grupo entero se desplaza y la variable temporal se coloca en la posición cero. Sólo debemos tener cuidado de no salirnos fuera de los límites del vector. Así, podemos asegurar que, cuando el bucle `for` más externo termina, el vector está ordenado.

4.4 Herencia múltiple

La *herencia múltiple* se emplea para derivar una clase a partir de varias clases base. Java no permite la herencia múltiple.

Todos los ejemplos de herencia vistos hasta ahora derivan una clase a partir de una única clase base. En la *herencia múltiple* una clase puede derivarse de varias clases base. Por ejemplo, supóngase que se tienen las clases `Estudiante` y `Trabajador`, un `EstudianteTrabajador` podría derivarse entonces de ambas clases.

Aunque la herencia múltiple es atractiva y en algunos lenguajes (incluyendo C++) se puede implementar, está rodeada de sutiles detalles que complican su diseño. Por ejemplo, es posible que las dos clases base contengan dos métodos con la misma signatura pero distintas implementaciones. Del mismo modo, ambas clases pueden tener atributos con nombres idénticos. Entonces, ¿cuál deberíamos usar?

Supongamos que en el ejemplo del `EstudianteTrabajador`, `Persona` es una clase con el atributo `nombre` y el método `toString`. Supóngase además que `Estudiante` extiende a `Persona` y sobrescribe el método `toString` para incluir el año de graduación del estudiante. Supongamos, también, que `Empleado` extiende a `Persona` pero no redefine `toString`, sino que lo declara como `final`.

1. Como `EstudianteTrabajador` hereda los atributos de `Estudiante` y `Trabajador`, ¿se tienen dos copias de `nombre`?
2. Si `EstudianteTrabajador` no redefine `toString`, ¿qué método `toString` debemos emplear?

Cuando en la herencia intervienen más clases, los problemas son aún mayores. Estas dificultades se traducen en implementaciones o identificadores de atributos conflictivos. Como consecuencia, Java no permite la herencia múltiple; sin embargo, presenta una alternativa llamada *interfaz*.

4.5 La interfaz

La *interfaz* es una clase abstracta que no contiene ningún detalle de implementación.

En Java la `interface` es la clase más abstracta posible. Consiste sólo en métodos públicos abstractos y atributos públicos y finales.

Se dice que una clase *implementa* una interfaz si define todos los métodos abstractos de la interfaz. Una clase que implementa una interfaz se comporta como si hubiese extendido una clase abstracta especificada por la interfaz.

En principio, la diferencia fundamental entre una interfaz y una clase abstracta es que, aunque ambas especifican cómo deben comportarse las subclases, la interfaz no puede dar ningún detalle acerca de la implementación de los métodos ni de los atributos. La consecuencia de estas restricciones es que la existencia de varias interfaces no presenta los mismos problemas potenciales que la herencia múltiple. De este modo, mientras que una clase puede extender sólo a otra clase base, sí que puede implementar a varias interfaces.

4.5.1 Especificación de una interfaz

Sintácticamente, nada es más sencillo que especificar una interfaz. La interfaz tiene la apariencia de una declaración de clase, salvo que emplea la palabra clave `interface`. Consiste en una lista de métodos que debe ser posteriormente implementados. Un ejemplo es la interfaz `Comparable`, mostrada en la Figura 4.12 y empleada varias veces a lo largo del texto.

La interfaz `Comparable` especifica dos métodos que cualquier subclase suya debe implementar: `compara` y `menorQue`. `compara` se comporta como el método `String compareTo`. Observe que no es necesario especificar que estos métodos son públicos (mediante `public`) y abstractos (mediante `abstract`). Como esto está forzado en los métodos de una interfaz, estas palabras claves pueden, y deben, ser omitidas.

```
1 package Soporte;
2
3 public interface Comparable
4 {
5     int compara( Comparable lder );      // Similar a compareTo
6     boolean menorQue( Comparable lder );
7 }
```

Figura 4.12 Interfaz `Comparable`.

4.5.2 Implementación de una interfaz

Una clase implementa una interfaz si cumple los siguientes requisitos:

1. declara que implementa la interfaz y
2. define implementaciones para todos los métodos de la interfaz.

La *cláusula* `implements` se emplea para indicar que una clase implementa una interfaz. Dicha clase debe implementar todos los métodos de la interfaz o, en caso contrario, ser abstracta.

En la Figura 4.13 se muestra un ejemplo. En él se define la clase `MiEntero`, empleada como clase estándar a lo largo del texto. La clase `MiEntero` se comporta de forma idéntica a la clase predefinida de Java `Integer`, pero además podrá usarse siempre que se requiera la manipulación de objetos `Comparables`.

En la línea 3, en la que la clase declara que implementa la interfaz, se usa la palabra clave `implements` en lugar de `extends`. La clase puede contener todos los métodos que desee, pero entre ellos deben figurar los listados en la interfaz. La interfaz se implementa entre las líneas 23 y 30. Nótese que es obligatorio implementar *exactamente cada método* indicado en la interfaz. Por ello, estos métodos tienen como párametro un objeto `Comparable` en lugar de un objeto `MiEntero`.

Una clase que implementa una interfaz puede ser extendida a su vez, siempre que no sea final. Así, como `MiEntero` no es final, podría ser extendida posteriormente.

```
1  package Soporte;
2
3  final public class MiEntero implements Comparable
4  {
5          // Constructor
6      public MiEntero( int x )
7        { valor = x; }
8
9          // Algunos métodos
10     public String toString( )
11       { return Integer.toString( valor ); }
12
13     public int intValor( )
14       { return valor; }
15
16     public boolean equals( Object lder )
17     {
18         return lder instanceof Integer &&
19                valor == ((MiEntero)lder).valor;
20     }
21
22         // Implementando la interfaz
23     public int compara( Comparable lder )
24     {
25         return valor < ((MiEntero)lder).valor? -1 :
26                valor == ((MiEntero)lder).valor? 0 : 1;
27     }
28
29     public boolean menorQue( Comparable lder )
30       { return valor < ((MiEntero)lder).valor; }
31
32         // Atributos
33     private int valor;
34 }
```

Figura 4.13 La clase `MiEntero` (versión preliminar), que implementa la interfaz `Comparable`.

Además, cualquier clase que implemente una interfaz puede extender también a otra clase. Por ejemplo, podríamos haber escrito

```
public class MiEntero extends Integer implements Comparable
```

Esto sería incorrecto, sólo porque `Integer` es una clase final que no puede extenderse.

```
1  package Soporte;
2
3  public interface Hashable
4  {
5      int hash( int tamanyoTabla );
6  }
```

Figura 4.14 La interfaz `Hashable`.

```
 1  package Soporte;
 2
 3  /**
 4   * Clase empleada en estructuras de datos genéricas.
 5   * Se comporta igual que Integer.
 6   */
 7  final public class MiEntero implements Comparable, Hashable
 8  {
 9      public MiEntero( int x )
10        { /* Como antes */ }
11      public String toString( )
12        { /* Como antes */ }
13      public int intValor( )
14        { /* Como antes */ }
15      public boolean equals( Object lder )
16        { /* Como antes */ }
17
18      public int compara( Comparable lder )
19      {
20          return valor < ((MiEntero)lder).valor? -1 :
21                 valor == ((MiEntero)lder).valor? 0 : 1;
22      }
23
24      public boolean menorQue( Comparable lder )
25      {
26          return valor < ((MiEntero)lder).valor;
27      }
28
29      public int hash( int tamanyoTabla )
30      {
31          if( valor < 0 )
32              return -valor % tamanyoTabla;
33          else
34              return valor % tamanyoTabla;
35      }
36
37      private int valor;   /* Como antes */
38  }
```

Figura 4.15 La clase `MiEntero`, que implementa dos interfaces.

4.5.3 Varias interfaces

Como ya hemos mencionado anteriormente, una clase puede implementar varias interfaces y la sintaxis para hacerlo es muy sencilla. Una clase implementa varias interfaces si cumple los siguientes requisitos:

1. enumera las interfaces que implementa y
2. define implementaciones para todos los métodos de dichas interfaces.

La Figura 4.14 muestra la interfaz `Hashable`, estudiada con detalle en la Sección 6.7 y el Capítulo 19. La clase `MiEntero` implementa las interfaces `Comparable` y `Hashable`. El resultado (en el que para mayor brevedad hemos eliminado los métodos que no corresponden a ninguna interfaz) se muestra en la Figura 4.15.

La interfaz es la clase más abstracta posible y una elegante solución al problema de la herencia múltiple.

4.6 Implementación de componentes genéricas

La *programación genérica* permite realizar implementaciones independientes del tipo de los objetos base.

Recordemos que un objetivo central de la programación orientada a objetos es la reutilización de código. Un mecanismo destacado que hace posible alcanzarlo es la programación *genérica*: cuando una implementación no depende del tipo de los objetos que manipula, puede emplearse una *implementación genérica* para describir la funcionalidad básica. Por ejemplo, si se pretende ordenar un vector de elementos, la lógica empleada para hacerlo es independiente del tipo de los objetos manipulados, luego puede emplearse un método genérico.

En Java, la genericidad se obtiene a través de la herencia.

A diferencia de muchos de los lenguajes más recientes (como C++, que emplea plantillas para implementar la programación genérica), Java no soporta directamente las implementaciones genéricas. Esto es debido a que la programación genérica puede implementarse a partir de la herencia. Esta sección describe cómo las clases y métodos genéricos pueden implementarse en Java empleando los conceptos básicos de la herencia[1].

En Java la idea básica es que se puede implementar una clase genérica utilizando una superclase adecuada, como `Object`. En Java, si una clase no extiende a ninguna otra se entiende que extiende a la clase `Object` (definida en `java.lang`). Como resultado, cualquier clase es una subclase de `Object`.

Considere la clase `CeldaEntera` de la Figura 3.1 y recuerde también que `CeldaEntero` soporta los métodos `leer` y `escribir`. Podemos convertir esta clase en una clase genérica llamada `CeldaMemoria`. El resultado se muestra en la Figura 4.16.

Las clases envoltorio deben emplearse para obtener genericidad con tipos primitivos.

Cuando se emplea esta estrategia debe tenerse en cuenta dos detalles. Ambos se ilustran en la Figura 4.17, en la que se implementa un método `main` que escribe un 5 en un objeto `CeldaMemoria` y después lee el contenido de dicho objeto. En primer lugar, debe recordarse que un tipo primitivo no es un objeto. Como resultado, `m.escritura(0)` sería incorrecto. Sin embargo, esto no representa un grave problema, ya que Java define *clases envoltorio* para los ocho tipos primitivos. Así, el objeto `CeldaMemoria` guardará un objeto de clase `Integer` (y no un valor de tipo `int`).

[1] El soporte directo de métodos y clases genéricos se encuentra en la actualidad bajo una fuerte consideración, pretendiéndose ampliar para ello el lenguaje con la palabra clave `generic`. Actualmente la técnica descrita en esta sección es la más utilizada.

```
1  // Clase CeldaMemoria
2  // Object leer( )          --> Devuelve el valor guardado
3  // void escribir( Object x)-> x es almacenado
4
5  public class CeldaMemoria
6  {
7          // Métodos públicos
8      public Object leer( )          { return valorGuardado; }
9      public void escribir( Object x){ valorAlmacenado = x; }
10
11         // Representación interna y privada de datos
12     private Object valorAlmacenado;
13 }
```

Figura 4.16 Clase genérica `CeldaMemoria`.

```
1  public class TestCeldaMemoria
2  {
3      public static void main( String [ ] args )
4      {
5          CeldaMemoria m = new CeldaMemoria( );
6
7          m.escribir( new Integer( 5 ) );
8          System.out.println( "El contenido es: " +
9                      ( (Integer) m.leer( ) ).intValue( ) );
10     }
11 }
```

Figura 4.17 Empleo de la clase genérica `CeldaMemoria`.

Suelen necesitarse las conversiones de tipos para obtener tipos del resultado genéricos

El segundo detalle es que el resultado de `m.leer( )` es un `Object`. Para generar el valor almacenado en cada momento, debemos realizar previamente una conversión de tipos, convirtiendo el `Object` en un `Integer`. Después, aplicaremos el método `intValue` para obtener un `int`[2].

Debido a que las clases envoltorio son clases finales, el constructor y el método de acceso `intValue` pueden ser expandidos por el compilador, obteniéndose código tan eficiente como si un `int` hubiera sido manipulado directamente.

Un segundo ejemplo es el problema de la ordenación. Ya hemos escrito anteriormente un método `ordenacionPorInsercion` que trabaja con un vector de `Figuras`. Ahora sería oportuno escribirlo para un vector genérico.

La Figura 4.18 muestra una rutina genérica de inserción que es casi idéntica al método de la Figura 4.9. Este método está incluido en la clase `Ordenacion` del paquete `EstructurasDatos`; posteriormente se irá añadiendo nuevas rutinas de ordenación a dicha clase, que será ampliamente estudiadas en el Capítulo 8 del libro. Es importante observar que en la rutina de ordenación no se emplea la clase `Object`. Por el contrario, ordenamos un vector de elementos `Comparables`, puesto que es necesario emplear el método `menorQue`. Como consecuencia, sólo las clases que implementan la interfaz `Comparable` pueden ser ordenadas con el nuevo `ordenacionPorInsercion`. Así, este método, tal y como está escrito,

[2] Realmente, no se necesita aplicar `intValue`, ya que el método `toString` está directamente definido en la clase `Integer`. Aquí lo empleamos para mostrar cómo el valor de tipo `int` podría obtenerse de una forma más general.

```
1  // ordenacionPorInsercion: ordenación del vector a.
2  // Forma parte de la clase EstructurasDatos.Ordenacion
3  // El paquete importa Soporte.*
4
5      private static void ordenacionPorInsercion( Comparable [ ] a )
6      {
7          for( int p = 1; p < a.length; p++ )
8          {
9              int j = p;
10             Comparable tmp = a[ p ];
11
12             for( ; j > 0 && tmp.menorQue( a[ j - 1 ] ); j-- )
13                 a[ j ] = a[ j - 1 ];
14             a[ j ] = tmp;
15         }
16     }
```

Figura 4.18 Rutina genérica de inserción.

no puede emplearse para la ordenación de un vector de `Figuras`, ya que la clase `Figura` no implementa la interfaz `Comparable`. El Ejercicio 4.15 pide al lector que realice las modificaciones necesarias a la clase `Figura` para poder hacerlo.

Para comprobar cómo se emplea el método genérico `Ordenacion` vamos a escribir un programa que lea una cantidad ilimitada de enteros, los ordene y muestre el resultado. El mismo se muestra en la Figura 4.19. El método `leerEnteros`, escrito en la Figura 2.4, se emplea para la lectura del vector de `int`s en la línea 11.

```
1  import Soporte.*;
2
3  // Programa de prueba para leer enteros del terminal (uno por
4  // línea), ordenarlos y finalmente mostrarlos.
5
6  public class OrdenaEnteros
7  {
8      public static void main( String [ ] args )
9      {
10             // Lectura del vector de enteros
11         int [ ] arr = LecturaEnteros.leerEnteros( ); // Figura 2.4
12
13             // Conversión a un vector de MiEnteros
14         MiEntero [ ] elVector = new MiEntero[ arr.length ];
15         for( int i = 0; i < elVector.length; i++ )
16             elVector[ i ] = new MiEntero( arr[ i ] );
17
18             // Aplicación del algoritmo de ordenación
19         EstructurasDatos.Ordenacion.ordenacionPorInsercion( elVector );
20
21             // Muestra el resultado ordenado
22         System.out.println( "Resultado ordenado: " );
23         for( int i = 0; i < elVector.length; i++ )
24             System.out.println( elVector[ i ] );
25     }
26 }
```

Figura 4.19 Lectura de un conjunto de `int`s, ordenación de los mismos y muestra del resultado.

A continuación, se genera un vector de elementos que implementen nuestra interfaz `Comparable`. La clase predefinida `Integer` no implementa `Comparable`, pero sí lo hace la clase `MiEntero`, presentada en este capítulo. En las líneas 14 a 16 se realiza la construcción apropiada. Para ello se crea por cada `int` el correspondiente `MiEntero`, que luego se guarda en el vector. La ordenación se efectúa en la línea 19 (no empleamos la cláusula `import`, por lo que el método debe ser invocado con su nombre completo, incluyendo los paquetes en los que está integrado). Por último, el código para mostrar el resultado se encuentra en las líneas 22 a 24. Observe que el método `toString` se invoca implícitamente a través de `MiEntero`.

Resumen

La herencia es una potente herramienta, esencial en la programación orientada a objetos y en Java. Nos permite abstraer funcionalidad dando lugar a las clases abstractas base e implementar clases derivadas expandiendo dicha funcionalidad. En la clase base pueden especificarse varios tipos de métodos, tal y como resume la Figura 4.20.

Método	Sobrecarga	Comentarios
final	En tiempo de compilación	Invariante a lo largo de la jerarquía de herencia (el método nunca se redefine).
abstract	En ejecución	La clase base no facilita ninguna implementación y es abstracta. Las clases derivadas deben facilitar la implementación.
static	En tiempo de compilación	No hay objeto controlador.
Otros	En ejecución	La clase base facilita una implementación que puede sobrescribirse o no en las clases derivadas.

Figura 4.20 Cuatro tipos de métodos.

Las clases más abstractas, en las que no se puede incorporar ninguna implementación, son las `interfaces`. Una interfaz enumera todos los métodos que deben ser implementados por cada clase derivada. Las clases derivadas deben implementar todos estos métodos, o ser, a su vez, abstractas, además de indicar, mediante la palabra clave *implements*, que implementan la interfaz. Una clase puede implementar varias interfaces, representando esto una alternativa sencilla a la herencia múltiple.

Finalmente, la herencia permite escribir de forma sencilla métodos y clases genéricas mediante las cuales representamos una amplia gama de tipos genéricos. Para lograrlo tendremos habitualmente que utilizar la conversión de tipos.

Con este capítulo concluye la primera parte del texto, en la que hemos visto las características básicas de Java y de la programación orientada a objetos. Pasaremos ahora a estudiar los algoritmos y las estructuras de datos empleados en la resolución de problemas de programación.

Elementos del juego

clase abstracta Clase de la que no pueden construirse objetos pero que sirve para especificar la funcionalidad de sus clases derivadas.

clase base Clase sobre la que se basa la herencia.

clase derivada Clase completamente nueva que presenta cierta compatibilidad con la clase de la que se deriva.

clase envoltorio Clase que facilita objetos en los que se almacenan elementos de un tipo primitivo. Un ejemplo de clase envoltorio es la clase `Integer`.

clase final Clase que no puede ser extendida.

clase hoja Clase final.

composición Mecanismo más adecuado para implementar la herencia cuando la relación subyacente no es una relación ES-UN(A). En tal caso se dice que un objeto de clase *B* contiene un objeto de clase *A*.

`extends` Cláusula empleada para indicar que una clase es una subclase de otra.

herencia Proceso gracias al cual podemos derivar una clase partiendo de otra, llamada base, sin modificar la implementación de la clase base. También permite el diseño de jerarquías de clases, como `Exception` y `Component`.

herencia múltiple Proceso en el que se deriva una clase a partir de varias clases base. Java no soporta la herencia múltiple; sin embargo sí se tiene la alternativa de implementar simultáneamente varias interfaces.

`implements` Cláusula necesaria para declarar que una clase implementa los métodos de una interfaz.

interfaz Tipo especial de clase abstracta que no contiene ningún detalle de implementación.

ligado (tipado) dinámico Decisión tomada en tiempo de ejecución en la que se determina qué metodo debe aplicarse al objeto referenciado en ese momento.

ligado (tipado) estático La decisión de qué método se aplica se toma en tiempo de compilación. Se emplea únicamente para métodos estáticos o finales.

llamada al constructor `super` Llamada al constructor de la clase base.

método abstracto Método especificado pero no definido en la clases abstractas, y que, por ello, tendrá que ser implementado en las clases derivadas.

método final Método que no puede sobrescribirse y, por tanto, permanece invariante a lo largo de la jerarquía de herencia. Para estos métodos se emplea el ligado (tipado) estático.

miembro de clase protegido Accesible para las clases derivadas pero privado para cualquier otra.

ordenación por inserción Algoritmo de ordenación sencillo, adecuado para entradas de pequeño tamaño.

polimorfismo Capacidad que poseen las variables referencia para apuntar a objetos de diferentes tipos. Cuando se realizan operaciones con una variable tal, se

selecciona automáticamente la adecuada al objeto referenciado en el momento actual.

programación genérica Se emplea para implementar la lógica independiente del tipado.

referencia `super` Referencia empleada en la sobrescritura (sobrecarga) parcial mediante la cual se invoca un método de la correspondiente clase base.

relación CONTIENE Relación en la que la clase derivada contiene una instancia de la correspondiente clase base.

relación ES-UN(A) Relación en la que la clase derivada es una variación de la clase base.

sobrescritura (sobrecarga) parcial Hecho de extender el cuerpo de un método de una clase base, para realizar tareas adicionales, aunque no completamente distintas.

Errores comunes

1. Las componentes privadas de una clase base no son accesibles desde sus clases derivadas.
2. Los objetos de una clase abstracta no puede instanciarse.
3. Si la clase derivada no implementa un método heredado de la clase abstracta, la primera se convierte también en abstracta. Si esto no es indicado explícitamente, el compilador generará un error.
4. Los métodos finales no deben ser sobrescritos. Del mismo modo, las clases finales no pueden extenderse.
5. Los métodos estáticos emplean el ligado estático, incluso si son redefinidos en una clase derivada.
6. En una clase derivada, las componentes heredadas sólo deben inicializarse a través del método `super`. Si estas componentes son públicas o protegidas, puede ser leídas o asignadas posteriormente.
7. En una clase derivada, la lista `throws` de un método sobrescrito no puede incluir ninguna excepción que no sea lanzada en el mismo método de la clase base. Además los tipos del resultado deben coindir en ambos métodos.
8. En la mayoría de los casos en los que un método genérico devuelve una referencia genérica, debe realizarse una conversión de tipos para obtener el objeto devuelto.

En Internet

El código para el caso de estudio `Figura` se encuentra en el directorio **Chapter04**. `Sort.java` es parte del paquete `DataStructures`, contenido en el directorio del mismo nombre. El código que ilustra el ejemplo sobre genericidad `CeldaMemoria` también se encuentra disponible en **Chapter04**. Por último, los interfaces de las Figuras 4.12, 4.14 y 4.15 pueden encontrarse en el directorio **Supporting**.

Circle.java	La clase `Circulo`.
Comparable.java	La interfaz `Comparable` de la Figura 4.12, formando parte del paquete `Soporte`.
Hashable.java	La interfaz `Hashable` de la Figura 4.14, formando parte del paquete `Soporte`.
MemoryCell.java	La clase `CeldaMemoria` de la Figura 4.16.
MyInteger.java	La interfaz `MiEntero` de la Figura 4.15, formando parte del paquete `Soporte`.
Rectangle.java	La clase `Rectangulo`.
Shape.java	La clase abstracta `Figura`.
Sort.java	Colección de rutinas genéricas de ordenación, se encuentran en el paquete `EstructurasDatos`.
SortInts.java	Contiene el código de la Figura 4.19 (clase `OrdenaEnteros`).
Square.java	La clase `Cuadrado`.
TestMemoryCell.java	Programa de prueba para la clase de celdas de memoria, mostrado en la figura 1.17 (clase `CeldaMemoria`).
TestShape.java	Un programa de prueba para el ejemplo del `Cuadrado`.
Underflow.java	La clase excepción `DesbordamientoInferior` de la Figura 4.3, formando parte del paquete `Soporte`.

Ejercicios

Cuestiones breves

4.1. ¿Qué componentes de una clase heredada pueden emplearse en la clase derivada? ¿Qué componentes se convierten en públicos para los usuarios de la clase derivada?

4.2. ¿Qué es la composición?

4.3. Explique el concepto de polimorfismo.

4.4. Explique el ligado (tipado) dinámico. ¿Cuándo no se emplea este tipo de ligado?

4.5. ¿Qué es un método final?

4.6. Considere el programa de la Figura 4.21, para estudiar la visibilidad.

a) ¿Qué accesos son incorrectos?

b) Si `main` fuese un método de `Base`, ¿qué accesos serían incorrectos?

c) Si `main` fuese un método de `Derivada`, ¿qué accesos serían incorrectos?

d) ¿Cómo cambiarían las respuestas anteriores si se eliminase de la línea 4 la cláusula `protected`?

e) Escriba un constructor de `Base` con tres parámetros. Escriba, también, un constructor de `Derivada` con cinco parámetros.

f) La clase `Derivada` tiene cinco atributos enteros. ¿Cuáles son accesibles para la clase `Test`?

g) Un método en la clase `Derivada` se aplica sobre un objeto de `Base`. ¿A qué componentes de `Base` puede acceder `Derivada`?

```
1  public class Base
2  {
3      public    int bPublico;
4      protected int bProtegido;
5      private   int bPrivado;
6      // Se omiten los métodos públicos
7  }
8
9  public class Derivada extends Base
10 {
11     public    int dPublico;
12     private   int dPrivado;
13     // Se omiten los métodos públicos
14 }
15
16 public class Test
17 {
18     public static void main( String [ ] args )
19     {
20         Base b     = new Base( );
21         Derivada d = new Derivada( );
22
23         System.out.println( b.bPublico + " " + b.bProtegido + " "
24                             + b.bPrivado + " " + d.dPublico + " "
25                             + d.dPrivado );
26     }
27 }
```

Figura 4.21 Programa para comprobar la visibilidad.

4.7. ¿Cuál es la diferencia entre las clases finales y el resto de clases? ¿Para qué se emplean las clases finales?

4.8. ¿Qué es un método abstracto?

4.9. ¿Qué es una clase abstracta?

4.10. ¿Qué es una interfaz? ¿En qué se diferencia una interfaz de una clase abstracta? ¿Cómo deben ser los componentes de una interfaz?

4.11. ¿Cómo se implementan en Java los algoritmos genéricos?

Problemas prácticos

4.12. Escriba dos métodos genéricos `min` y `max`. Cada uno de ellos debe tener dos parámetros de tipo `Comparable`. Emplee ambos métodos sobre la clase `MiEntero`.

4.13. Escriba dos métodos genéricos `min` y `max` que tengan como parámetro de entrada un vector de `Comparables`. Emplee ambos métodos sobre la clase `MiEntero`.

4.14. Modifique en el ejemplo de la clase `Figura` los métodos `leerFigura` y `main`, de modo que lancen y recogan una excepción (en lugar de crear un círculo de radio cero), cuando se detecte un error en la entrada.

4.15. Modifique la clase `Figura` para que el algoritmo genérico de ordenación pueda emplearse para manipular objetos de ese tipo.

4.16. Un `BufferSimple` soporta las operaciones `lectura` y `escritura`: dicho `BufferSimple` almacena un solo valor, de modo que un atributo suyo nos indica si está lleno o vacío. Una operación de `escritura` sólo podrá efectuarse cuando el `BufferSimple` esté vacío, mientras que las operaciones de `lectura` se ejecutan cuando haya un elemento almacenado. Una `lectura` borra el contenido del `BufferSimple` y devuelve el valor de lo que almacenaba justo antes. Escriba una clase genérica `BufferSimple`, definiendo las excepciones necesarias para la detección de errores.

Prácticas de programación

4.17. Rescriba la jerarquía de `Figura` de modo que se calcule el área de los objetos en el constructor de la misma y se almacene en un atributo de la clase. Los constructores de las clases derivadas deben pasar el resultado de ese cálculo al método `super`. El método `area` debe ser final y devolver únicamente el valor de ese atributo.

4.18. Añada a la jerarquía de `Figura` el concepto de posición, incluyendo como atributos las coordenadas necesarias. Añada, después, un método `distancia`.

4.19. Escriba una clase abstracta para manipular `Fechas` y su clase derivada `FechaGregoriana`.

4.20. Implemente una jerarquía de contribuyentes que consista en una interfaz `Contribuyente` y en las clases `ContribuyenteSoltero` y `ContribuyenteCasado` que implementan dicha interfaz.

Bibliografía

Los siguientes libros describen los principios generales del desarrollo del software orientado a objetos:

1. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ (1988).
2. B. Booch, *Object-Oriented Design and Analysis with Applications*, 2.ª ed., Benjamin/Cummings, Redwood City, Calif. (1994).
3. I. Jacobson, M. Christerson, P. Jonsson, y G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach* (revisado), Addison-Wesley, Reading, Mass. (1992).
4. D. de Champeaux, D. Lea, y P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, Mass. (1993).