Empezar a Programar Usando Java

Natividad Prieto
Francisco Marqués
Isabel Galiano
Jorge González
Carlos Martínez-Hinarejos
Javier Piris
Assumpció Casanova
Marisa Llorens
Jon Ander Gómez
Carlos Herrero
Germán Moltó

Capítulo 13

Extensión del comportamiento de una clase. Herencia

Como se ha mencionado en el capítulo 2, uno de los objetivos fundamentales de la POO es la de facilitar la *reutilización del código*. Ello permite volver a emplear elementos que al haber sido ya realizados son bien conocidos y están, posiblemente, exhaustivamente probados. En particular, en los lenguajes de programación orientados a objetos el mecanismo básico para el reuso del código es la *herencia*. Mediante ella es posible definir nuevas clases extendiendo o restringiendo las funcionalidades de otras clases ya existentes.

La *herencia* es un mecanismo que permite modelar relaciones jerárquicas entre elementos, del tipo *is a* (*es un(a)*), por ejemplo, esta es la relación que se da entre una máquina y un ordenador, en la que un ordenador *es una* máquina. En una relación así un elemento, el *heredero*, tiene las características de otro elemento pero, tal vez, refinándolas para definirlo como un caso especial del primero. Nótese que, para el ejemplo anterior, si un `Ordenador` *es una* `Maquina`, también un `PCCompatible` *es un* `Ordenador`, así como `miPC` es, a su vez, un `PCCompatible`. Naturalmente, desde la POO `Maquina`, `Ordenador` y `PCCompatible` son todos ellos clases, que forman una *jerarquía*, siendo una instancia de todos ellos (un objeto) `miPC`.

Desde el punto de vista del lenguaje Java hay dos puntos donde la herencia es particularmente relevante. Por una parte, la herencia se emplea exhaustivamente en el propio lenguaje a lo largo del conjunto de librerías de clases que posee. Por otra parte el lenguaje, como cabía prever, da soporte a la definición de nuevas clases *herederas* de las características de otras ya definidas. Cualquier clase, predefinida o definida por el programador, es una clase que hereda de `Object`, la clase base de la jerarquía.

13.1 Jerarquía de clases. Clases base y derivadas

La herencia es el mecanismo mediante el cual se utiliza la definición de una clase llamada *base* para definir una nueva clase llamada *derivada*, la cual hereda sus propiedades. La relación de herencia entre clases genera lo que se denomina *jerarquía*. A modo de ejemplo, se plantea definir dos clases: `Circulo` y `Rectangulo`. Supóngase que ambas clases definen un atributo para la posición y otro para el color; además, en `Circulo` se define el radio y en `Rectangulo` la base y la altura. La funcionalidad de las clases se define a través de los métodos *gets* y *sets* correspondientes y otros para desplazar la figura o para calcular su área. En la tabla 13.1 se muestra de forma resumida esta información.

Clase	Atributos y Métodos

`Circulo`	`Point2D.Double posicion;`
	`Color color;`
	`double radio;`

	`Point2D.Double getPosicion()`
	`Color getColor()`
	`double getRadio()`
	`void setPosicion(double, double)`
	`void setColor(Color)`
	`void setRadio(double)`
	`void desplazar(double)`
	`double area()`

`Rectangulo`	`Point2D.Double posicion;`
	`Color color;`
	`double base, altura;`

	`Point2D.Double getPosicion()`
	`Color getColor()`
	`double getBase()`
	`double getAltura()`
	`void setPosicion(double, double)`
	`void setColor(Color)`
	`void setBase(double)`
	`void setAltura(double)`
	`void desplazar(double)`
	`double area()`

Tabla 13.1: Las clases `Circulo` y `Rectangulo`.

Ambas clases comparten elementos comunes (los atributos `posicion`, `color` y los métodos `getPosicion()`, `getColor()`, `setPosicion(double, double)`, `setColor(Color)` y `desplazar(double)`) con los que podría definirse una nueva clase `Figura`, y tanto `Circulo` como `Rectangulo` se definirían como clases que derivan de ella. Esta jerarquía puede ampliarse. Por ejemplo, se puede definir una nueva clase `Cuadrado` como una derivada de `Rectangulo` con la base y la altura

iguales. La relación que se establece entre las clases es una relación de herencia ya que tanto `Circulo` como `Rectangulo` heredan de `Figura` parte de sus atributos y métodos. La herencia es una relación transitiva, es decir, `Cuadrado` hereda los atributos y operaciones de `Rectangulo` junto a las que éste hereda de `Figura`. La relación entre las cuatro clases se ilustra en la figura 13.1, donde las flechas indican que una clase es una derivada de una clase base: `Figura` es la clase base de la que derivan tanto `Circulo` como `Rectangulo` y `Rectangulo` es la clase base de la que deriva `Cuadrado`.

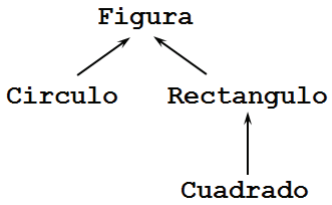

Figura 13.1: Ejemplo de jerarquía de clases.

La herencia establece una relación *es una* entre las clases, (`NomClaseDerivada` *es una* `NomClaseBase`). En la tabla 13.2 se muestra la relación de herencia y los atributos propios y heredados de cada clase.

Clase	Relación	Atributos propios y heredados
`Figura`	clase base	`posicion, color`
`Circulo`	*es una* `Figura`	`posicion, color radio`
`Rectangulo`	*es una* `Figura`	`posicion, color base, altura`
`Cuadrado`	*es un* `Rectangulo`	`posicion, color base, altura`

Tabla 13.2: Relación de herencia y atributos.

Como ya se ha comentado, la relación de herencia es intrínseca a la propia definición del lenguaje, dando lugar a una *jerarquía de clases*. La base de toda la jerarquía es la clase `Object`, de la que derivan todas las clases. En la figura 13.2 se ilustra parte de la jerarquía de algunas componentes para el diseño de interfaces gráficas. En esta jerarquía los objetos de tipo `JFrame` son visualizables en pantalla y pueden contener componentes gráficas distribuidas en contenedores como `JPanel`. Otros ejemplos que se verán en los siguientes capítulos son la jerarquía `Throwable` en el capítulo 14 y las jerarquías `InputStream` y `Writer` en el capítulo 15.

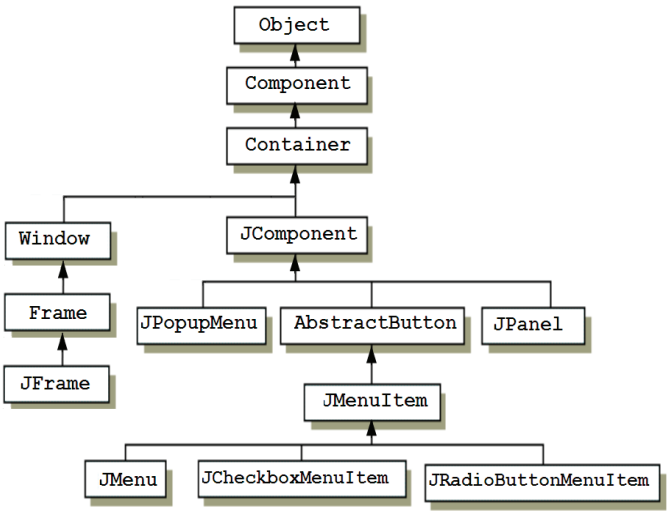

Figura 13.2: Ejemplo de jerarquía de las librerías de Java.

La jerarquía de clases es una estructura lógica basada en la relación de herencia. Existe otra jerarquía basada en la ubicación de los ficheros en los que se definen las clases. Estos ficheros se organizan en directorios dando lugar también a una estructura de árbol conocida como *jerarquía de librerías o paquetes* que se describirá con detalle en la sección 13.5. Los paquetes contienen clases y subpaquetes con funcionalidades fuertemente relacionadas entre sí. En un programa Java se accede a los paquetes separando sus nombres por puntos. Por ejemplo, `java.awt.geom.Point2D.Double` representa el camino para acceder a la clase `Point2D.Double` que se encuentra en la librería `geom` localizada en la librería `awt` del lenguaje Java. Todas las clases del paquete `java.awt.geom` están diseñadas para trabajar con figuras geométricas en el plano. Otro ejemplo son las clases del paquete `javax.swing` que definen componentes gráficas como `JFrame` y `JPanel` usadas para diseñar interfaces gráficas de usuario.

13.2 Diseño de las clases base y derivadas: `extends`, `protected` y `super`

En esta sección se abordan los mecanismos que aporta Java para definir la relación de herencia. Las clases derivadas se implementan usando la palabra reservada `extends`. La sintaxis para crear una clase derivada es:

```
[modificadores] class NomClaseDerivada extends NomClaseBase { ... }
```

Se dice que la clase `NomClaseDerivada` es una clase *derivada*, *descendiente* o que *es hija* o *subclase* de la clase `NomClaseBase` y que la clase `NomClaseBase` es la *clase base*, la *clase padre* o *superclase* de la clase `NomClaseDerivada`; o simplemente que `NomClaseDerivada` *extiende* `NomClaseBase`. Desde un punto de vista conceptual, también se dice que `NomClaseDerivada` es una *especialización* de `NomClaseBase`.

En el ejemplo de las figuras, la clase `Figura` se puede definir como la Clase Tipo de Dato de la figura 13.3.

En `Figura` se definen los atributos: `color`, de tipo predefinido `Color`, y `posicion`, la posición de la figura en el plano, de tipo también predefinido `Point2D.Double`. Estas dos clases están definidas en la librería `java.awt` y `java.awt.geom`, respectivamente. En `Color` se definen constantes públicas de clase para algunos colores estándar como `Color.red` (rojo), `Color.green` (verde), etc. La representación interna de cada color en la clase `Color` es una combinación de los tres colores básicos rojo, verde y azul (*rgb*) en cantidades de tipo `byte`. Por ejemplo: la combinación `[r=255,g=0,b=0]` representa el color rojo. La clase `Point2D.Double` define una coordenada (x, y) en el plano. Los métodos comunes a todas las figuras son los que se definen en la clase `Figura`, por ejemplo, los consultores `getColor()` y `getPosicion()`. Nótese que no existe ningún método para el cálculo del área de una figura. Esto se debe a que el área se calcula de forma distinta para cada figura y, por tanto, se implementa con un método distinto en cada clase derivada. Usando el mecanismo de herencia, se puede extender la clase `Figura` con las peculiaridades de cada figura:

```
public class Circulo extends Figura { ... }
public class Rectangulo extends Figura { ... }
```

Modificador `protected`

Como se explicó en su momento, los atributos de una clase deben ser privados, es decir, visibles sólo en la clase en la que se definen, y accesibles sólo a través de métodos públicos *gets* y *sets*. Esto implementa el *principio de ocultación* que aporta la ventaja de que cualquier cambio en la estructura de datos repercute sólo en la clase en la que se realizan los cambios facilitando así el mantenimiento de la aplicación. Si se respeta esto, las clases derivadas no tienen acceso directo a los atributos privados de la clase base aunque debido a la relación existente, en ocasiones, resulte apropiado permitir el acceso. Se podría resolver este problema declarando los atributos públicos dejando la estructura de datos al descubierto; de esta forma, cualquier modificación en la estructura implica cambios en el resto del programa. No obstante, la solución adecuada es hacer que las componentes de la clase base sean visibles sólo por las clases derivadas, es decir, expandir el principio de ocultación a las clases derivadas. Para ello se usa la palabra reservada `protected`. Así, existen cuatro tipos de visibilidad para los elementos definidos en

```
import java.awt.Color;
import java.awt.geom.Point2D;
/**
 *  Clase Figura: representa una figura geométrica,
 *  con la funcionalidad que aparece a continuación.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class Figura {
    protected Color color;
    protected Point2D.Double posicion;

    /** Crea una Figura de color c y centro en (x, y).
     *  @param c Color, el color.
     *  @param x double, la abscisa del centro.
     *  @param y double, la ordenada del centro.
     */
    public Figura(Color c, double x, double y) {
        this.color = c; posicion = new Point2D.Double(x, y);
    }

    /** Devuelve el color de la Figura.
     *  @return Color, el color.
     */
    public Color getColor() { return color; }

    /** Devuelve la posición del centro de la Figura.
     *  @return Point2D.Double, la posición del centro.
     */
    public Point2D.Double getPosicion() { return posicion; }

    /** Actualiza el color de la Figura a nuevoC.
     *  @param nuevoC Color, el nuevo color.
     */
    public void setColor(Color nuevoC) { color = nuevoC; }

    /** Actualiza la posición del centro de la Figura a (pX, pY).
     *  @param pX double, la nueva abscisa del centro.
     *  @param pY double, la nueva ordenada del centro.
     */
    public void setPosicion(double pX, double pY) {
        posicion.setLocation(pX, pY);
    }

    /** Desplaza d unidades la posición del centro de la Figura.
     *  @param d double, las unidades a desplazar.
     */
    public void desplazar(double d) {
        posicion.setLocation(posicion.getX() + d, posicion.getY() + d);
    }

    /** Devuelve un String con el color y la posición de la Figura.
     *  @return String, representación de los datos de la Figura.
     */
    public String toString() {
        return ", color: " + color + " y\n\t posición: " + posicion;
    }
}
```

Figura 13.3: Clase `Figura` de la jerarquía de figuras.

una clase: `public`, `protected`, `private` y *friendly*. Esta última es la visibilidad que se aplica cuando no se utiliza ninguna de las anteriores. La siguiente tabla muestra el alcance de la visibilidad de un elemento de una clase según el modificador usado en su definición, suponiendo que todas las clases están en el mismo paquete.

Visible en	`private`	`protected`	`public`	*friendly*
la propia clase	sí	sí	sí	sí
una clase derivada	no	sí	sí	sí
una clase no derivada	no	no	sí	sí

Conceptualmente, si los atributos de la clase base se definen `private` se expresa que un objeto de la clase base es una *subparte* de la clase derivada y que esa *subestructura* de datos sólo es accesible con métodos *gets* y *sets*. El uso del modificador `protected` permite heredar los atributos como si los atributos de la clase base estuvieran definidos en la propia clase derivada formando parte de la misma estructura sin la necesidad de invocar a los métodos *gets* y *sets*.

Creación de objetos y `super`

Se puede invocar al constructor de la clase base en la definición del constructor de una clase derivada. Para invocar al constructor de la clase base, se usa la palabra reservada `super` sin nombrar explícitamente la clase base. La sintaxis de esta invocación es:

$$\texttt{super(arg}_1\texttt{, arg}_2\texttt{, ..., arg}_n\texttt{);}$$

en donde $\texttt{arg}_1$, $\texttt{arg}_2$, ..., $\texttt{arg}_n$ forman la lista de argumentos, siendo n el número de parámetros del constructor de la clase base ($n \geq 0$).

El lenguaje impone una restricción: cuando se invoca al constructor de la clase base en el constructor de la clase derivada, ésta debe ser la primera instrucción del cuerpo del método. En este caso, el código para definir la clase `Circulo` queda como sigue:

```
public class Circulo extends Figura {
    private double radio;

    public Circulo(double r, Color c, double x, double y) {
        super(c, x, y);
        this.radio = r;
    }
    ...
}
```

Es recomendable usar `super` desde el punto de vista del mantenimiento y de la reutilización de código. Además de invocar un código ya escrito, si se cambia la estructura de datos de la clase base, la clase derivada será mas resistente a los cambios ya que estos sólo afectarán al constructor en la clase base.

Si un constructor de una clase derivada no invoca explícitamente a un constructor de la clase base, el compilador de Java inserta automáticamente una llamada al constructor sin argumentos de la clase base (`super();`). Si la clase base no tiene un constructor sin argumentos, ocurrirá un error en tiempo de compilación. La clase `Object` tiene dicho constructor, por lo que si `Object` es la única clase base de una clase derivada, no hay ningún problema.

En las figuras 13.4, 13.5 y 13.6 se muestra la implementación de todas las clases que conforman la jerarquía de figuras junto con la propia `Figura`.

Sobrescritura de métodos heredados y `super`

Al igual que los atributos, las clases derivadas también heredan los métodos no privados de la clase base y por lo tanto también se pueden aplicar a objetos de la clase derivada. Por ejemplo, se puede obtener el color y la posición de un círculo sin haber definido los métodos `getColor()` y `getPosicion()` en la clase `Circulo` como ocurre en el siguiente ejemplo:

```
Circulo c = new Circulo(4.5, Color.blue, 3.0, 5.0);
Color col = c.getColor();
Point2D.Double p = c.getPosicion();
```

Existen ocasiones en las que se desea cambiar la implementación de algún método heredado para particularizar su comportamiento en la clase derivada. Cuando en la clase derivada se define un método con el mismo perfil que el de la clase base, se anula la herencia de dicho método. En este caso se dice que el método de la clase derivada *sobrescribe* al de la clase base. Si se desea invocar el método sobrescrito de la clase base, se usa la palabra `super`. A continuación, se presenta un ejemplo de sobrescritura en el que se utiliza `super` para llamar al método homónimo de la clase base.

Ejemplo 13.1. La clase `Corona` de la figura 13.7 representa una corona circular formada por dos círculos concéntricos. `Corona` puede definirse como una derivada de la clase `Circulo` con un atributo `radioIn` para el radio del círculo interior ya que el centro y el color son los mismos que los del círculo exterior. En esta clase, el constructor crea un círculo externo invocando al constructor de la clase padre con el mayor de los radios e inicializa el radio interno con el menor. El área de una corona se obtiene restando el área del círculo interno a la del círculo externo. Para obtener el área del círculo externo se invoca al método `area()` de la clase base,

```
import java.awt.Color;
import java.util.Locale;
/**
 * Clase Circulo: define un círculo de un determinado radio, color y
 * posición de su centro, con la funcionalidad que aparece a continuación.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class Circulo extends Figura {
    protected double radio;

    /** Crea un Circulo de radio r, color c y centro en (x, y).
     *  @param r double, el radio.
     *  @param c Color, el color.
     *  @param x double, la abscisa del centro.
     *  @param y double, la ordenada del centro.
     */
    public Circulo(double r, Color c, double x, double y) {
        super(c, x, y); this.radio = r;
    }

    /** Devuelve el radio del Circulo.
     *  @return double, el radio.
     */
    public double getRadio() { return radio; }

    /** Actualiza el radio del Circulo a nuevoR.
     *  @param nuevoR double, el nuevo radio.
     */
    public void setRadio(double nuevoR) { radio = nuevoR; }

    /** Devuelve el área del Circulo.
     *  @return double, el área.
     */
    public double area() { return Math.PI * radio * radio; }

    /** Devuelve un String con los datos del Circulo.
     *  @return String, representación de los datos del Circulo.
     */
    public String toString() {
        String res = "Círculo de radio ";
        String formato = String.format(Locale.US, "%.2f", radio);
        res += formato + super.toString();
        return res;
    }
}
```

Figura 13.4: Clase `Circulo` de la jerarquía de figuras.

```
import java.awt.Color;
import java.util.Locale;
/**
 * Clase Rectangulo: define un rectángulo dados su base, altura, color y
 * posición de su centro, con la funcionalidad que aparece a continuación.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class Rectangulo extends Figura {
    protected double base, altura;

    /** Crea un Rectangulo de base b, altura a, color c y centro en (x, y).
     *  @param b double, la base.
     *  @param a double, la altura.
     *  @param c Color, el color.
     *  @param x double, la abscisa del centro.
     *  @param y double, la ordenada del centro.
     */
    public Rectangulo(double b, double a, Color c, double x, double y) {
        super(c, x, y); this.base = b; this.altura = a;
    }

    /** Devuelve la base del Rectangulo.
     *  @return double, la base.
     */
    public double getBase() { return base; }

    /** Devuelve la altura del Rectangulo.
     *  @return double, la altura.
     */
    public double getAltura() { return altura; }

    /** Actualiza la base del Rectangulo a nuevaB.
     *  @param nuevaB double, la nueva base.
     */
    public void setBase(double nuevaB) { base = nuevaB; }

    /** Actualiza la altura del Rectangulo a nuevaA.
     *  @param nuevaA double, la nueva altura.
     */
    public void setAltura(double nuevaA) { altura = nuevaA; }

    /** Devuelve el área del Rectangulo.
     *  @return double, el área.
     */
    public double area() { return base * altura; }

    /** Devuelve un String con los datos del Rectangulo.
     *  @return String, representación de los datos del Rectangulo.
     */
    public String toString() {
        String res = "Rectángulo de ";
        String formato = String.format(Locale.US,
            "base %.2f,altura %.2f", base, altura);
        res += formato + super.toString();
        return res;
    }
}
```

Figura 13.5: Clase `Rectangulo` de la jerarquía de figuras.

```
import java.awt.Color;
import java.util.Locale;
/**
 * Clase Cuadrado: define un cuadrado de un determinado lado, color y
 * posición de su centro, con la funcionalidad que aparece a continuación.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class Cuadrado extends Rectangulo {
    /** Crea un Cuadrado de lado lado, color c y centro en (x, y).
     *  @param lado double, el lado.
     *  @param c Color, el color.
     *  @param x double, la abscisa del centro.
     *  @param y double, la ordenada del centro.
     */
    public Cuadrado(double lado, Color c, double x, double y) {
        super(lado, lado, c, x, y);
    }

    /** Devuelve el lado del Cuadrado.
     *  @return double, el lado.
     */
    public double getLado() { return base; }

    /** Actualiza el lado del Cuadrado a nuevoL.
     *  @param nuevoL double, el nuevo lado.
     */
    public void setLado(double nuevoL) {
        base = nuevoL; altura = nuevoL;
    }

    /** Devuelve un String con los datos del Cuadrado.
     *  @return String, representación de los datos del Cuadrado.
     */
    public String toString() {
        String res = "Cuadrado de lado ";
        String formato = String.format(Locale.US, "%.2f", base);
        res += formato + ", color: " + color;
        res += " y\n\t posición: " + posicion;
        return res;
    }
}
```

Figura 13.6: Clase `Cuadrado` de la jerarquía de figuras.

y la del círculo interno se obtiene creando un nuevo círculo de radio `radioIn` y aplicando el método `area()` de la clase `Circulo`. El atributo `posicion` es del tipo predefinido `Point2D.Double` que proporciona los métodos `getX()` y `getY()` para obtener el valor en la abscisa y en la ordenada, respectivamente.

La sobrescritura de métodos es muy habitual en Java. Como ya se vió en la sección 4.5.2, los métodos `toString()` y `equals(Object)`, definidos en la clase `Object`, se sobrescriben para adecuarlos a cada clase. Es el caso del método `toString()` de la clase `Figura` (en la figura 13.3) que se sobreescribe en cada una de sus clases derivadas, como se puede observar en las figuras 13.4, 13.5 y 13.6. En la sección 13.3.2 se verá con detalle la sobrescritura del método `toString()` en la jerarquía de figuras. A continuación, se presenta un ejemplo de sobrescritura del método `paintComponent(Graphics)` definido en la clase `JComponent`.

Ejemplo 13.2. El método `paintComponent(Graphics)` se invoca automáticamente cada vez que se requiere dibujar una componente gráfica en pantalla usando un objeto de tipo `Graphics`, el cual contiene toda la información necesaria para ello. Esto ocurre cuando una componente gráfica aparece en primer plano de la pantalla o cambia de tamaño o forma. Este método se define en la clase `JComponent` y lo heredan todas sus derivadas, entre ellas la clase `JPanel` base de la clase `MiPanel` de la figura 13.8. Normalmente, este método dibuja la componente con un fondo en blanco. La nueva funcionalidad del método en esta última clase consiste en añadir al `JPanel` una imagen de fondo guardada en el fichero *fondo.jpg* y escalarla cuando el método se llame para redibujar.

La primera instrucción invoca a `paintComponent(Graphics)` de la clase base para dibujar la componente sin imagen de fondo. Después se obtienen las dimensiones de `MiPanel` y se escala la imagen a las nuevas dimensiones de la ventana. La clase `ImageIcon` se usa para crear un icono con la imagen guardada en el fichero. El método `getImage()` devuelve la imagen del icono de la que se obtiene la nueva imagen escalada pasando las nuevas dimensiones a los dos primeros parámetros del método `getScaledInstance(int, int, int)` de la clase `Image` del paquete `java.awt`. La nueva imagen se inserta en un icono para dibujarla.

Las instrucciones que siguen crean una ventana (`JFrame`) a la que se le asigna unas dimensiones iniciales y, tras añadirle el objeto de la clase `MiPanel`, la dibuja en pantalla invocando al método `setVisible(boolean)`. En la figura 13.9 se aprecia el efecto del escalado del fondo al cambiar la forma de la ventana.

```
JFrame v = new JFrame();
v.add(new MiPanel("fondo.jpg"));
v.setSize(100, 100);
v.setVisible(true);
```

```
import java.awt.Color;
/**
 * Clase Corona: define una corona circular dados sus radios
 * externo e interno, color y posición de su centro, con la
 * funcionalidad que aparece a continuación.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class Corona extends Circulo {
    private double radioIn;

    /** Crea una Corona de radios radio1 y radio2,
     *  color c y centro en (x, y).
     *  @param radio1 double, un radio.
     *  @param radio2 double, otro radio
     *  @param c Color, el color.
     *  @param x double, la abscisa del centro.
     *  @param y double, la ordenada del centro.
     */
    public Corona(double radio1, double radio2,
                  Color c, double x, double y) {
        super(Math.max(radio1, radio2), c, x, y);
        this.radioIn = Math.min(radio1, radio2);
    }

    /** Devuelve el área de la Corona.
     *  @return double, el área.
     */
    public double area() {
        Circulo c = new Circulo(radioIn, color,
            posicion.getX(), posicion.getY());
        double areaIn = c.area();
        return super.area() - areaIn;
    }
}
```

Figura 13.7: Clase `Corona` de la jerarquía de figuras.

```
import java.awt.Graphics;
import java.awt.Image;
import javax.swing.ImageIcon;
import javax.swing.JPanel;
/**
 * Clase MiPanel: representa un panel con una imagen de fondo.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class MiPanel extends JPanel {
    protected ImageIcon fondo;

    /** Crea un MiPanel con la imagen del fichero nomFich.
     *  @param nomFich String, el nombre del fichero.
     */
    public MiPanel(String nomFich) {
        fondo = new ImageIcon(nomFich);
    }

    /** Dibuja una componente gráfica en pantalla
     *  con una imagen de fondo.
     *  @param g Graphics, la componente gráfica.
     */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int alturaJPanel = this.getHeight();
        int anchoJPanel = this.getWidth();
        Image imagXEscalar = fondo.getImage();
        Image imagEscalada = imagXEscalar.getScaledInstance(
            anchoJPanel, alturaJPanel, Image.SCALE_FAST);
        ImageIcon nuevoFondo = new ImageIcon(imagEscalada);
        nuevoFondo.paintIcon(this, g, 0, 0);
    }
}
```

Figura 13.8: Sobrescritura del método `paintComponent(Graphics)`.

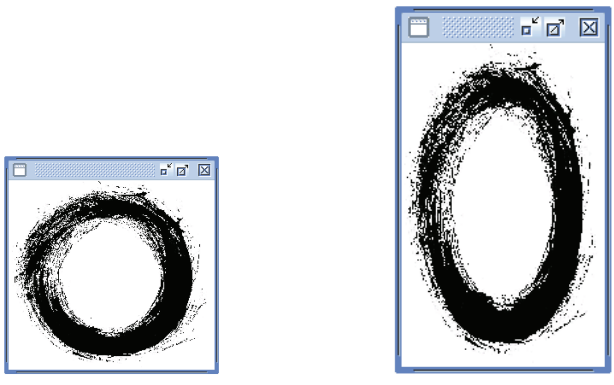

Figura 13.9: Ejemplo de escalado.

13.3 Uso de una jerarquía de clases. Polimorfismo

En la secciones 3.3 y 3.4 se estudió el concepto de compatibilidad de tipos para los tipos básicos y los tipos referencia. En esta sección se extiende este concepto para el caso de los tipos referencia que forman parte de una jerarquía. En presencia de herencia, se dice que dos tipos son compatibles si pertenecen a la misma línea de la jerarquía en sentido descendente. La compatibilidad garantiza que toda variable de tipo referencia `C` pueda referenciar a un objeto de la clase `C` y sus derivadas, pero no a la inversa. De esta forma, la herencia proporciona un *polimorfismo de variables* al poder referenciar a una variedad de tipos de objetos. En Java el siguiente código es totalmente legal, siendo las variables `f1` y `f2` capaces de referenciar tanto a un `Circulo` como a un `Rectangulo`:

```
Figura f1 = new Circulo(4.5, Color.blue, 3.0, 5.0);
Figura f2 = new Rectangulo(4.5, 3.2, Color.red, 3.0, 4.0);
```

La herencia también proporciona un *polimorfismo de métodos* pudiendo aplicarse los métodos heredados de la clase base a objetos instancia de todas sus clases derivadas. Como ocurre en el siguiente código:

```
Color cf1 = f1.getColor();
Color cf2 = f2.getColor();
```

donde el mismo método `getColor()`, heredado de la clase `Figura`, se aplica a dos objetos de distinto tipo.

13.3.1 Tipos estáticos y dinámicos

Cuando en Java se define una variable referencia, se le asocia el tipo de los objetos a los que puede referenciar. Este tipo es conocido en tiempo de compilación y se conoce como *tipo estático*. En tiempo de compilación se comprueba si se producen violaciones de tipos estáticos. Los lenguajes con tipos estáticos pueden manejar tipos explícitos o tipos inferidos. El primer caso consiste en declarar el tipo de la variable sin importar que sea un atributo o una variable de clase. Por ejemplo:

```
// definición del tipo estático de un atributo
Figura a;
// definición del tipo estático de una variable de clase
static Figura b;
```

En el segundo caso, el compilador infiere los tipos de las expresiones y las declaraciones de acuerdo al contexto. Por ejemplo, ante la expresión

```
Double d = new Double(2.0) + 3;
```

el compilador deduce que el tipo del número `3` de tipo `int` debe ser `Double` y le aplica una conversión implícita de tipo.

Además de un tipo estático, las variables también poseen un tipo *dinámico* que se obtiene durante la ejecución. El tipo dinámico de una variable es el tipo del objeto al que referencia la variable. Por ejemplo, en la expresión

```
Object o = new Double(3.5);
```

el tipo estático de la variable `o` es `Object` y el tipo dinámico es `Double`.

Durante la ejecución, el tipo dinámico se maneja mediante un *enlace dinámico*. Cuando se aplica un método a un objeto, se determina la clase a la que pertenece el objeto, y se busca el método correspondiente en la clase. En el caso de no encontrarse, se busca en la clase base ascendiendo en la jerarquía. Este proceso se repite hasta encontrar la clase donde se define el método que se desea ejecutar. Si se llega al final de la jerarquía sin encontrarlo, se produce un error en la ejecución. Por ejemplo, considerando la jerarquía:

```
java.lang.Object
  |
  +--Figura
        |
        +--Circulo
```

y que la variable `f1` referencia a un círculo, la instrucción `f1.getColor()` ejecuta el método definido en la clase `Figura` ya que no está definido en la clase `Circulo` pero sí en la clase `Figura`. La instrucción `f1.equals(f2)` ejecuta el método definido en la clase `Object` ya que no está definido en la clase `Circulo` ni en `Figura`.

El enlace dinámico permite al programador usar la sobrescritura como un regulador de la herencia: sabiendo que el enlace dinámico se aplicará igualmente, el programador decide sobrescribir o no un método heredado y, por tanto, modificar o mantener invariante un comportamiento definido en la clase base.

13.3.2 Ejemplo de uso del polimorfismo

En las secciones anteriores se ha visto que la herencia aporta polimorfismo a las variables y métodos dinámicos, y que junto al principio de ocultación, permite un

mantenimiento más simple del software. En esta sección se plantea un ejemplo de su uso. Considerando la jerarquía de las figuras, se plantea el diseño de una clase `GrupoFiguras` para guardar varios tipos de figuras y poder gestionarlo añadiendo o eliminando una figura, calculando el área de todas ellas, mostrando su información en forma de cadena de caracteres, etc. El principal objetivo consiste en diseñar la clase `GrupoFiguras` de forma que el código quede inalterado ante una posible modificación del conjunto de tipos de figuras. Centrando el ejemplo en el diseño de un método para mostrar su información en forma de cadena de caracteres, se debe sobrescribir el método `toString()`, definido en la clase `Object`, en todas las clases de la jerarquía de `Figura`.

El método `toString()` en la clase `Object`

En la clase `Object`, este método devuelve la información básica de un objeto sobre una cadena de caracteres de la forma `NombreClase@direcciónMemoria`, donde `NombreClase` es el tipo dinámico y `direcciónMemoria` es una representación en hexadecimal de la primera posición de memoria que ocupa el objeto en el montículo (heap). Por ejemplo, si no se sobrescribe el método `toString()` en la clase `Figura` ni en ninguna de sus clases derivadas, las siguientes instrucciones:

```
Figura f = new Figura();
System.out.println(f);
```

muestran por la salida estándar:

```
Figura@50618d26
```

Lo mismo ocurre al aplicar el método `toString()` a los objetos de las clases derivadas de `Figura` ya que, al no sobrescribir el método heredado, ejecutan el método de la clase `Object`.

El método `toString()` en la jerarquía `Figura`

En las figuras 13.3, 13.4, 13.5 y 13.6 se puede ver la sobrescritura del método `toString()` de `Object` en cada una de las clases de la jerarquía de figuras.

Cabe señalar que al concatenar una cadena con un objeto, se invoca automáticamente el método `toString()` de la clase del objeto sin necesidad de explicitarlo. Es decir, las expresiones `"" + posicion` y `"" + color` son equivalentes a escribir `"" + posicion.toString()` y `"" + color.toString()`, donde el método `toString()` es el definido en las clases `Point2D.Double` y `java.awt.Color`, respectivamente.

Los métodos de las clases `Circulo` y `Rectangulo` invocan al método de su clase base (`Figura`) usando `super`. Nótese que el método `toString()` de la clase `Cuadrado` no invoca al de su clase base para evitar que aparezca la palabra *Rectángulo*. A continuación se muestra el efecto de la sobrescritura en las clases `Cuadrado` y `Circulo`.

```
Figura f1 = new Circulo(4.5, Color.blue, 3.0, 5.0);
Figura f2 = new Cuadrado(2.0, Color.red, 3.0, 3.0);
System.out.println(f1 + "\n" + f2);
```

El resultado de la ejecución de las instrucciones anteriores en la salida estándar será el que sigue:

Entrada/Salida Estándar

```
Círculo de radio 4.50, color: java.awt.Color[r=0,g=0,b=255] y
         posición: Point2D.Double[3.0, 5.0]
Cuadrado de lado 2.00, color: java.awt.Color[r=255,g=0,b=0] y
         posición: Point2D.Double[3.0, 3.0]
```

El método `toString()` en la clase `GrupoFiguras`

A continuación se abordan tres aproximaciones para implementar la clase `GrupoFiguras` y su método `toString()`: usando varios arrays, usando el tipo `Object` y usando el tipo `Figura`. En cada una de ellas se avanza hacia un código más estable ante modificaciones del conjunto de figuras.

Primera aproximación: sin usar herencia

Una aproximación simple sin usar herencia consiste en definir un array de tamaño `MAX` por cada tipo de figura, una operación para insertar cada tipo de figura en el array correspondiente y, finalmente, sobrescribir el método `toString()` implementando un recorrido sobre cada uno de los arrays, como se muestra en la figura 13.10.

Con esta aproximación, por cada nuevo tipo de figura se debe añadir un nuevo array, un nuevo contador, un nuevo método `insertar` y un nuevo bucle en el método `toString()`. Nótese que no se mantiene información sobre el orden de inserción.

```
/**
 * Clase GrupoFiguras1: representación de un conjunto de figuras.
 * Primera aproximación: sin usar herencia.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class GrupoFiguras1 {
    private static final int MAX = 100;
    private Circulo[] circulos;
    private Rectangulo[] rectangulos;
    private Cuadrado[] cuadrados;
    private int numCirculos, numRectangulos, numCuadrados;

    /** Crea un GrupoFiguras1 sin figuras. */
    public GrupoFiguras1() {
        circulos = new Circulo[MAX];
        rectangulos = new Rectangulo[MAX];
        cuadrados = new Cuadrado[MAX];
        numCirculos = 0; numRectangulos = 0; numCuadrados = 0;
    }

    /** Añade un circulo c al grupo de figuras.
     *  @param c Circulo, el círculo.
     */
    public void insertar(Circulo c) {
        if (numCirculos < MAX) { circulos[numCirculos++] = c; }
    }

    /** Añade un rectangulo r al grupo de figuras.
     *  @param r Rectangulo, el rectángulo.
     */
    public void insertar(Rectangulo r) {
        if (numRectangulos < MAX) { rectangulos[numRectangulos++] = r; }
    }

    /** Añade un cuadrado c al grupo de figuras.
     *  @param c Cuadrado, el cuadrado.
     */
    public void insertar(Cuadrado c) {
        if (numCirculos < MAX) { cuadrados[numCuadrados++] = c; }
    }

    /** Devuelve un String con la información de todas
     *  las figuras del grupo.
     *  @return String, representación de las figuras.
     */
    public String toString() {
        String res = "";
        for (int i = 0; i < numCirculos; i++) {
            res += circulos[i] + "\n";
        }
        for (int i = 0; i < numRectangulos; i++) {
            res += rectangulos[i] + "\n";
        }
        for (int i = 0; i < numCuadrados; i++) {
            res += cuadrados[i] + "\n";
        }
        return res;
    }
}
```

Figura 13.10: Clase `GrupoFiguras1`. Primera aproximación: sin usar herencia.

Segunda aproximación: usando el tipo `Object`

Una primera forma de evitar los cambios anteriores, consiste en usar el tipo polimórfico `Object` y definir un único array y un único método `insertar(Object)`, como se muestra en la figura 13.11.

```
/**
 * Clase GrupoFiguras2: representación de un conjunto de figuras.
 * Segunda aproximación: usando el tipo Object.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class GrupoFiguras2 {
    private static final int MAX = 100;
    private Object[] objetos;
    private int numObjetos;

    /** Crea un GrupoFiguras2 sin figuras. */
    public GrupoFiguras2() {
        objetos = new Object[MAX]; numObjetos = 0;
    }

    /** Añade un Object o al grupo de figuras.
     *  @param o Object, el objeto.
     */
    public void insertar(Object o) {
        if (numObjetos < MAX) { objetos[numObjetos++] = o; }
    }

    /** Devuelve un String con la información de todas
     *  las figuras del grupo.
     *  @return String, representación de las figuras.
     */
    public String toString() {
        String res = "";
        for (int i = 0; i < numObjetos; i++) { res += objetos[i] + "\n"; }
        return res;
    }
}
```

Figura 13.11: Clase `GrupoFiguras2`. Segunda aproximación: usando el tipo `Object`.

Con esta aproximación, se obtiene un código más conciso y menos sensible a cambios en la aplicación. Esta solución parece suficiente, sin embargo, no carece de problemas. Se presupone que el parámetro del método `insertar(Object)` sólo recibe referencias a los tipos descendientes de `Figura`. En realidad, el tipo `Object` permite que el método reciba referencias a cualquier objeto. Una forma de controlarlo consiste en verificar que el tipo del parámetro `o` es una de las figuras. Esto se comprueba utilizando la instrucción `instanceof` y en el caso de que no se trate de una de las figuras, no se produce inserción alguna.

```
    /** Añade un Object o al grupo de figuras. */
    public void insertar(Object o) {
        if (o instanceof Circulo
            || o instanceof Rectangulo || o instanceof Cuadrado) {
            if (numObjetos < MAX) { listaObjetos[numObjetos++] = o; }
        }
    }
```

Esta implementación también carece de independencia, ya que cada nuevo tipo de figura requiere cambios en la condición del primer `if`.

Tercera aproximación: usando herencia

Haciendo uso de la herencia, se restringe el tipo de la lista y el parámetro del método `insertar(Object)` al tipo `Figura`, como se muestra en la figura 13.12. De esta forma, el parámetro de `insertar(Figura)` puede referenciar a la variedad de tipos de objetos descendientes de `Figura`, entre los que se encuentran `Circulo`, `Rectangulo` y `Cuadrado`.

```
/**
 * Clase GrupoFiguras3: representación de un conjunto de figuras.
 * Tercera aproximación: usando herencia.
 * @author Libro IIP-PRG
 * @version 2016
 */
public class GrupoFiguras3 {
    private static final int MAX = 100;
    private Figura[] figuras;
    private int numFiguras;

    /** Crea un GrupoFiguras3 sin figuras. */
    public GrupoFiguras3() {
        figuras = new Figura[MAX]; numFiguras = 0;
    }

    /** Añade una Figura f al grupo de figuras.
     *  @param f Figura, la figura.
     */
    public void insertar(Figura f) {
        if (numFiguras < MAX) { figuras[numFiguras++] = f; }
    }

    /** Devuelve un String con la información de todas
     *  las figuras del grupo.
     *  @return String, representación de las figuras.
     */
    public String toString() {
        String res = "";
        for (int i = 0; i < numFiguras; i++) { res += figuras[i] + "\n"; }
        return res;
    }
}
```

Figura 13.12: Clase `GrupoFiguras3`. Tercera aproximación: usando herencia.

Con esta aproximación se aprovechan las ventajas que comporta el uso de la herencia, las variables polimórficas y la sobrescritura para que el código sea independiente del conjunto de tipos de figuras. Cuando se desee trabajar con nuevos tipos de figura, bastará con definir una nueva clase descendiente de la clase `Figura`.

No obstante, esta solución tampoco es completamente satisfactoria ya que permite almacenar objetos de tipo `Figura`:

```
GrupoFigura g = new GrupoFigura();
g.insertar(new Figura());
```

El problema reside en que la clase `Figura` se define como una clase auxiliar para definir las propiedades comunes a todas las figuras, pero no se pretende que exista un objeto de esta clase.

Otra carencia de las dos últimas aproximaciones, es la imposibilidad de realizar el enlace dinámico ya que todos los objetos del array son de tipo `Object` o de tipo `Figura`. Por ejemplo, si además de implementar el método `toString()`, se intenta calcular el área de todas las figuras, se requiere el uso de casting para localizar el método que calcula el área de cada tipo de figura y localizar los métodos `area()` de cada clase:

```
/** Devuelve el área de todas las figuras del grupo de figuras. */
public double area() {
    double areaTotal = 0;
    for (int i = 0; i < numObjetos; i++) {
        if (objetos[i] instanceof Circulo) {
            areaTotal += ((Circulo) objetos[i]).area();
        }
        if (objetos[i] instanceof Rectangulo) {
            areaTotal += ((Rectangulo) objetos[i]).area();
        }
        if (objetos[i] instanceof Cuadrado) {
            areaTotal += ((Cuadrado) objetos[i]).area();
        }
    }
    return areaTotal;
}
```

Como se verá más adelante, para evitar el uso de castings y la instrucción `instanceof`, se pueden definir clases en las que no se permite crear instancias y en las que se pueden definir métodos cuya implementación se aplaza a sus clases derivadas.

13.4 Más herencia en Java: control de la sobrescritura

Como ya se ha señalado en las secciones precedentes, en el diseño de una jerarquía de clases se puede emplear la *sobrescritura* para que las clases derivadas especialicen su comportamiento en lugar de definir métodos nuevos. De este modo se pueden aprovechar al máximo todas las ventajas que el uso de las variables polimórficas comporta. Ahora bien, con los mecanismos estudiados hasta el momento para implementar la herencia, el diseñador no posee más que un control parcial de la sobrescritura, pues ni puede impedir que se produzca ni puede forzar u obligar a realizarla. Para conseguir prohibir o imponer la sobrescritura, y así convertir a la herencia en un verdadero instrumento de reutilización del software, se requieren los modificadores Java `final` y `abstract`, respectivamente; a su presentación y estudio están dedicadas las siguientes subsecciones.

13.4.1 Métodos y clases finales

Como es sabido, una clase `Derivada` puede especializar a su clase `Base` cambiando el significado de los métodos heredados mediante sobrescritura. Ahora bien, si se desea que un método de la superclase, `f`, permanezca *invariante* en la jerarquía y, por tanto, prohibir su sobrescritura, dicho método se debe definir como *final* añadiendo a su cabecera de definición el modificador `final`. Así, cualquier intento de sobrescribir este método en una derivada provocará un error de compilación `f in Derivada cannot override f in Base; overridden method is final`.

También un atributo y una clase Java pueden ser finales; por ejemplo,

- el atributo `PI` de `Math` es *final*, pues representa a la constante π;
- todas las clases envoltorio de los tipos primitivos de Java (`Integer`, `Boolean`, `Character`, `Double`, etc.) y la clase `String`, ubicadas en el paquete `java.lang` son *finales*. Obsérvese en la documentación de cualquiera de estas clases que una clase final se ubica como una hoja del árbol de herencia Java para expresar que *no puede ser extendida* o reutilizada vía herencia y que, ya a nivel puramente sintáctico, basta poner el modificador `final` en su cabecera de definición para que automáticamente todas sus componentes sean definidas como finales.

El uso de componentes `final` no sólo evita su redefinición accidental sino que también permite generar código más eficiente puesto que el intérprete Java resuelve su función asignada en tiempo de compilación (estáticamente), al igual que con una componente `static`, y no en tiempo de ejecución (dinámicamente).

13.4.2 Métodos y clases abstractos

En una jerarquía de clases algunos métodos pueden cambiar su significado mediante sobrescritura mientras que otros, los finales, permanecen invariantes. Una situación intermedia es aquella en la que se desea compaginar la herencia forzosa de `final` con la especialización de la sobrescritura. Por ejemplo, parece razonable que el método `area()` forme parte de la especificación de la clase `Figura` y, por lo tanto, sea heredado por todas sus subclases; pero sólo existen fórmulas de cálculo del área específicas para cada tipo de `Figura`, por lo que el área de una `Figura` sólo se puede calcular en las derivadas `Circulo`, `Rectangulo`, etc.

Para imponer que todas las subclases de una superclase dada hereden un determinado método pero, al mismo tiempo, que cada subclase lo especialice, dicho método se debe definir como *abstracto*. Para ello,

1. En la implementación de la superclase se define un método cuya cabecera contiene el modificador `abstract` y sin cuerpo, es decir, donde el bloque determinado por `{}` se substituye por `;`. Por ejemplo, la definición de `area` en `Figura` sería:

   ```
   public abstract double area();
   ```

2. El método definido en la superclase debe ser implementado en todas sus subclases, esto es, definido en cada subclase con la misma cabecera de la superclase pero con un cuerpo ad-hoc, en lugar del `;`

Una clase que tiene al menos un método abstracto es una *clase abstracta* y así se debe declarar; por ejemplo, `Figura` es abstracta, por lo que se define:

```
public abstract class Figura { ... }
```

Una clase abstracta no puede ser instanciada, esto significa que no se pueden crear objetos de la clase vía operador `new`, aunque sí se pueden declarar variables referencia. Por ejemplo, si la clase `Figura` es abstracta, la creación de un objeto vía el operador `new` (`Figura fEstandar = new Figura();`), provoca el error de compilación `Figura is abstract - cannot be instantiated`; sin embargo, no produce error la creación de un objeto de tipo `Circulo` a través de una referencia de tipo `Figura` (`Figura c = new Circulo();`).

Una subclase debe implementar todos los métodos abstractos de la superclase, al menos que ésta a su vez sea una clase abstracta; si no lo hace, el compilador lo detectará y dará un error. Por ejemplo, si `Circulo` no implementa `area`, se produce el siguiente mensaje de error `Circulo is not abstract and does not override abstract method area() in Figura`.

La clase `Figura` modificada como abstracta quedaría como sigue:

```
import java.awt.Color;
import java.awt.geom.Point2D;
/**
 *  Clase Figura: clase abstracta para representar figuras geométricas.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public abstract class Figura {
    protected String color;
    protected Point2D.Double posicion;

    /** Crea una Figura de color c y centro en (x, y).
     *  @param c Color, el color.
     *  @param x double, la abscisa del centro.
     *  @param y double, la ordenada del centro.
     */
    public Figura(Color color, double x, double y) {
        this.color = color; posición = new Point2D.Double(x, y);
    }

    /** Devuelve el color de la Figura.
     *  @return Color, el color.
     */
    public String getColor() { return color; }

    /** Devuelve la posición del centro de la Figura.
     *  @return Point2D.Double, la posición del centro.
     */
    public Point2D.Double getPosicion() { return posicion; }

    /** Devuelve el área de la Figura.
     *  @return double, el área.
     */
    public abstract double area();

    /** Devuelve un String con el color y la posición de la Figura.
     *  @return String, representación de los datos de la Figura.
     */
    public String toString() {
        return ", color " + color + " y posición: " + posicion;
    }
}
```

Así se pueden insertar varios tipos de figura en un grupo con la seguridad de que todas implementan su método `area` y que en el grupo de figuras sólo se insertan

objetos de las clases derivadas de `Figura`. El cálculo del área total de todas las figuras se implementa con el método `area()` de la clase `GrupoFiguras`.

```
/** Devuelve el área de todas las figuras del grupo de figuras. */
public double area() {
    double areaTotal = 0.0;
    for (int i = 0; i < numFiguras; i++) {
        // se invoca al método de cada figura
        areaTotal += figuras[i].area();
    }
    return areaTotal;
}
```

La ejecución de la siguiente secuencia de instrucciones nos daría el área total del grupo de figuras.

```
GrupoFiguras g = new GrupoFiguras();
g.insertar(new Circulo(4.5, Color.blue, 3.0, 5.0));
g.insertar(new Rectangulo(4,6, Color.blue, 3.0, 4.0));
g.insertar(new Cuadrado(4.5, Color.blue, 3.0, 5.0));
// se invoca al método area de GrupoFiguras
System.out.println(g.area());
```

13.4.3 Interfaces y herencia múltiple

Una clase Java cuyos métodos son todos abstractos recibe el nombre específico de *interfaz* y se utiliza para describir un comportamiento o funcionalidad específica pero sin implementación. Una interfaz se caracteriza porque sus atributos sólo pueden ser públicos y finales, no posee métodos constructores al carecer de estructura y, finalmente, sus métodos son públicos y han de ser implementados obligatoriamente en cualquiera de sus derivadas -por lo que se dice que éstas implementan la interfaz en lugar de extenderla.

Los pasos para definir una clase interfaz `I` en Java, y para indicar que otra clase `D` la implementa, son los siguientes:

- En la cabecera de `I` se utiliza la palabra `interface` en lugar de `class`.
- En la cabecera de cada método de `I` no se deben escribir los modificadores `public` y `abstract` pues lo son por definición de interfaz.
- En la cabecera de la clase `D` que implementa `I` debe incluirse `implements I`:

  ```
  public class D implements I { ... }
  ```

  En el caso en el que la clase `D` extienda a `B`:

  ```
  public class D extends B implements I { ... }
  ```

  Si `D` implementa varias interfaces, sus nombres se separan por comas.

Es muy importante señalar ahora que, gracias a que una clase puede implementar tantas interfaces como sea necesario, la interfaz es el mecanismo con el que Java implementa la *herencia múltiple*, esto es, el hecho de que una clase derivada herede de más de una clase base distinta de `Object`.

- En el cuerpo de la clase `D` que implementa `I` se deben sobrescribir obligatoriamente todos los métodos definidos en `I`, salvo si D es una clase abstracta o interfaz, pues de lo contrario el intérprete Java lo advierte.

Algunas interfaces predefinidas en Java son `Cloneable`, `Comparable`, `Runnable` y `Serializable`.

13.5 Organización de las clases en Java

13.5.1 La librería de clases del Java

Un aspecto importante del lenguaje Java es la *librería de clases predefinidas*, ya que incluye un grupo muy amplio de tipos y operaciones relacionados con problemas comunes de la programación actual. Así, la librería de clases del Java permite tratar aspectos tan distintos como el tratamiento numérico, la gestión fiable de comunicaciones remotas y la realización de interfaces de usuario gráficas, entre otros muchos.

La librería de clases del lenguaje se encuentra organizada de forma jerárquica, teniendo como clase base la denominada `Object`. Así, todas las clases del Java son, de una forma u otra, descendientes de la clase `Object` y, por ello, heredan, a veces sobrescribiéndolos, los métodos de dicha superclase.

La herencia entre clases predefinidas es muy habitual en Java, pudiéndose representar la jerarquía de clases del lenguaje mediante un árbol de bastante profundidad. Véase, por ejemplo, la jerarquía correspondiente a la clase `JFrame`, tal y como aparece en la documentación del lenguaje:

```
java.lang.Object
  |
  +--java.awt.Component
       |
       +--java.awt.Container
            |
            +--java.awt.Window
                 |
                 +--java.awt.Frame
                      |
                      +--javax.swing.JFrame
```

Cada una de las clases del ejemplo: `Object`, `Component`, `Container`, `Window`, `Frame` y `JFrame`, heredan en sus definiciones los atributos y métodos de las clases precedentes, sobrescribiéndolos cuando así lo necesitan, de forma que las funcionalidades de una clase quedan definidas por las de las clases que extienden, junto con las aportadas por ella misma.

Nótese que los nombres de las clases del ejemplo anterior vienen antepuestos por las etiquetas `java.lang`, `java.awt` o `javax.swing`. Dichas etiquetas no son nombres de clases sino que son un mecanismo de agregación de clases, propio del Java, denominado *paquete* (`package`).

Toda la información referente a la jerarquía en la que se define una clase se puede consultar en la documentación que el propio lenguaje genera. En la figura 13.13 se muestra la referida a la clase `Double`; en la primera línea se nombra el paquete donde está definida la clase seguido del nombre de la clase. A continuación se dibuja el segmento de la jerarquía de clases que lleva desde `Object` hasta la clase `Double`, y en la última línea aparecen los interfaces que implementa. En concreto, la clase `Double` hereda de la clase `Number` e implementa la interfaz `Serializable` entre otras.

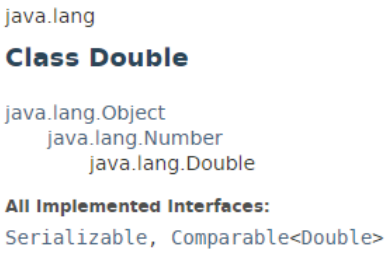

Figura 13.13: Documentación de la clase `Double`.

Seguido de esta información se muestran las siguientes tablas, que pueden aparecer o no en función de que la clase tenga o no elementos visibles desde fuera de la clase:

Field Summary con información sobre los atributos públicos y protegidos (`protected`) ya sean variables de instancia o de clase (`static`).

Fields inherited from class ... con los atributos heredados de otras clases; se muestra una tabla por cada clase de la que se heredan atributos.

Constructor Summary con los constructores de la clase.

Method Summary con una breve descripción de los métodos accesibles.

Methods inherited from class ... con métodos heredados de otras clases; se muestra una tabla por cada clase de la que se heredan métodos.

Methods inherited from class java.lang.Object se muestran los métodos heredados de la clase `Object`.

13.5.2 Uso de `packages`

En Java los paquetes (`packages`) se utilizan para poder agrupar un conjunto de clases que tienen funcionalidades comunes. Un `package` es, por lo tanto, un mecanismo del lenguaje para facilitar la organización de un conjunto de clases. Existen un grupo de paquetes predefinidos en los que se encuentran englobados todas las clases de la librería del lenguaje. Ejemplos de paquetes son los ya mencionados en el ejemplo anterior: `java.lang` y `java.awt`.

Cuando en un fichero con instrucciones Java se desea hacer uso de una clase de un paquete, es posible hacerlo usando el nombre completo prefijado de la clase, esto es el nombre del paquete seguido del de la clase, por ejemplo se puede hacer referencia a la clase `Frame` siempre que se necesite mediante `java.awt.Frame`. Sin embargo, suele ser más simple utilizar una directiva de importación de los elementos que interesen, como por ejemplo:

```
import java.awt.*;
// Se importan todas las clases del paquete java.awt

import java.awt.Frame;
// Se importa sólo la clase java.awt.Frame
```

De entre los paquetes y clases predefinidos del lenguaje Java cabe destacar por su importancia los siguientes:

- `java.applet`: definición y gestión de las *applets*, pequeñas aplicaciones que se descargan usando la `WWW` y se ejecutan, de forma segura, remotamente.
- `java.awt`: es el "Abstract Window Toolkit" o herramientas de ventana abstracta. Se trata de un conjunto de clases para la realización de interfaces de usuario gráficos, independientes de la plataforma específica en que se utilice.
- `java.io`: paquete que contiene las clases relativas a la entrada/salida en Java. Modela aspectos tan importantes como los flujos (`Stream`) y los ficheros (`File`).
- `java.lang`: es el paquete que contiene la mayoría de las clases de uso más común, como por ejemplo:
  - `Math`: funciones matemáticas como logarítmicas y trigonométricas.
  - `Object`: la clase raíz en la jerarquía de clases.
  - `String`: tratamiento de cadenas de caracteres.
  - `System`: aspectos básicos de gestión del sistema.
  - `Thread`: tratamiento para concurrencia y ejecución en paralelo.

- `java.util`: incluye un grupo de clases predefinidas, muchas de ellas tipos de datos de uso común, como por ejemplo: Pilas, Colas, Colecciones, Tablas Hash, Diccionarios, etc. La clase `Date`, para manejo de fechas y horas, también se encuentra entre ellas.
- `java.net`: contiene clases relativas a aplicaciones que acceden a redes.

El lenguaje permite construir nuevos paquetes agrupando distintas clases añadiendo la cláusula `package nombrePaquete` al principio del fichero que contiene la clase. Éstas podrán ser utilizadas posteriormente haciendo uso de los mecanismos de referenciación ya vistos. Por ejemplo, para que la jerarquía de figuras vista a lo largo del capítulo forme parte de un paquete denominado `figuras`[1], bastará con agrupar todas las clases de la jerarquía en un directorio con el mismo nombre y añadir, como primera línea de los ficheros que contienen cada una de las clases de la jerarquía, la instrucción:

```
package figuras;
```

Para utilizar una clase `public` del paquete `figuras`, por ejemplo, la clase `Circulo`, desde otra clase que no forme parte de dicho paquete, por ejemplo, la clase `SecuenciaDeCirculos` (figura 9.18), se puede importar solo la clase `Circulo`, añadiendo una instrucción `import` al comienzo (después de la instrucción `package`, si existe) del fichero de la clase `SecuenciaDeCirculos`:

```
import figuras.Circulo;
```

con lo que, en dicha clase, se podrá utilizar directamente el nombre para hacer referencia a la clase `Circulo`.

La organización de las clases en paquetes tiene muchas ventajas: permite organizar el código desarrollado, facilita la posibilidad de compartir código con otros programadores al eliminar la ambigüedad en los nombres de las clases y permite el uso del denominado *modificador de visibilidad de paquete o de acceso amistoso*, que es el que sintácticamente se aplica en ausencia de modificador de visibilidad (`public`, `private` o `protected`) y que permite el acceso desde clases que formen parte del mismo paquete. Por eso en Java todas las clases se ubican en paquetes, bien en el que se indica explícitamente, bien en un paquete `anonymous` en el que se agrupan todas las clases que no están definidas en un paquete determinado.

Siguiendo las recomendaciones del propio lenguaje sólo se utilizarán clases no ubicadas en paquetes cuando se desarrollen pequeñas aplicaciones o aplicaciones de prueba o bien en los comienzos del desarrollo de programas.

[1] En los capítulos que siguen, se supondrá que la jerarquía de figuras forma parte del paquete `figuras`.

13.6 Problemas propuestos

1. Tomando como modelo las clases `Circulo` y `Rectangulo` vistas en el capítulo, implementar una clase `Triangulo` que descienda de la clase `Figura`.

2. Añadir un método `perimetro` en la clase `Figura` de forma que todas las figuras tengan que implementarlo.

3. Especializar el método `perimetro` del ejercicio anterior para todas las clases que desciendan de la clase `Figura`.

4. En la clase `GrupoFiguras`:

   - Diseñar un método que calcule la suma de los perímetros de todas las figuras.
   - Diseñar un método que cree y devuelva un cuadrado con el área total de todas las figuras del grupo.
   - Diseñar un método que ordene las figuras en orden ascendente de su área.

5. Diseñar una clase `Poligono` similar a las clases `Circulo` y `Rectangulo` que guarde un número máximo de puntos dados en su constructor.

6. Rediseñar las clases `Circulo`, `Rectangulo` y `Triangulo` extendiendo la clase `Poligono`.

7. Si no se pueden crear objetos de una clase abstracta vía operador `new`, ¿qué utilidad tienen los métodos constructores de las clases abstractas? ¿Qué consecuencias tiene la siguiente definición en la clase `Figura`?

```
private Color color;
private Point2D.Double posicion;
public Figura(Color color, int x, int y) {
    this.color = color;
    posicion = new Point2D.Double(x, y);
}
```

8. Dada la siguiente jerarquía de clases en el paquete `vehiculos`:

```
public class Vehiculo {
    ...
    public Vehiculo(int potencia) { ... }
    public int potencia () { ... }
    ...
}
```

```
public class Coche extends Vehiculo {
    ...
    public Coche(int potencia, int numPlazas) { ... }
    public int numPlazas() { ... }
    ...
}

public class Moto extends Vehiculo {
    ...
    public Moto(int potencia) { ... }
    ...
}
```

Diseñar en este mismo paquete una clase `Garaje` de forma que:

- En el constructor se indique el número total de plazas del `Garaje`.
- En cada plaza se pueda guardar tanto un `Coche` como una `Moto`.
- Tenga una función que devuelva la cuota mensual de una plaza calculada de la forma siguiente:
  - si en dicha plaza hay un `Coche`, la cuota es la potencia multiplicada por el número de plazas;
  - si en la plaza hay una `Moto`, la cuota se calcula como la potencia multiplicada por 2;
  - si no hay ningún vehículo en la plaza, la cuota es 0.

9. Se dispone de las siguientes clases en el paquete `losAnimales`:

```
public class Milpies {
    protected int numeroDePies;
    public Milpies() {
        numeroDePies = 1000;
        escribirPies();
    }

    public void escribirPies() {
        System.out.println("Un Milpiés o Cochinilla tiene "
            + numeroDePies + " pies");
    }
}

public class MilpiesEsquiador extends Milpies {
    protected int numeroDePiesRotos;
    public MilpiesEsquiador() {
        numeroDePiesRotos = 100;
    }
```

```
    public void escribirPies() {
        System.out.println("A un Milpiés esquiador le quedan "
            + (numeroDePies - numeroDePiesRotos) + " pies");
    }
}

public class TestMilpies {
    public static void main(String[] args) {
        MilpiesEsquiador m = new MilpiesEsquiador();
    }
}
```

Indicar el motivo por el que el resultado de la ejecución del `main` de `TestMilpies` es `A un Milpiés esquiador le quedan 1000 pies`. Explicar también si el uso de `final` conseguiría cambiar el actual resultado de `TestMilpies`.

10. Sean las siguientes clases del paquete `losAnimales`:

```
public class Animal {
    public void emitirSonido() { System.out.println("Grunt"); }
}

public class Muflon extends Animal {
    public void emitirSonido() { System.out.println("MOOOO!"); }
    public void alimentarCon() { System.out.println("Hierba!"); }
}

public class Armadillo extends Animal { }

public class Guepardo extends Animal {
    public void emitirSonido() { System.out.println("Groar!"); }
}
```

- Si el siguiente programa Java se ubica también en el paquete `losAnimales`, indicar las instrucciones de su `main` que provocan error y las que no y explicar brevemente el motivo.

```
public class Test1Animal {
    public static void main(String[] args) {
        adoptar(new Armadillo());
        Object o = new Armadillo();
        Armadillo a1 = new Animal();
        Armadillo a2 = new Muflon();
    }

    private static void adoptar(Animal a) {
        System.out.println("Ven, cachorrito!");
    }
}
```

- Tracear el resultado de la ejecución del siguiente programa:

```
package losAnimales;
public class Test2Animal {
    public static void main(String[] args) {
        Animal a = new Armadillo();
        a.emitirSonido();
        a = new Muflon();
        a.emitirSonido();
        a = new Guepardo();
        a.emitirSonido();
    }
}
```

  En función del resultado obtenido y siguiendo las reglas de la herencia, ¿qué modificaciones se deberían realizar en la jerarquía para exigir que todos los animales emitan el sonido `Grunt`? ¿Cómo se podría conseguir saber el tipo de alimentación de cada animal?

11. Modificar la jerarquía `Animal` para garantizar que cada clase derivada de `Animal` defina el método `alimentarCon()` y con ello conocer el tipo de alimentación de cada `Animal`.

Más Información

[Eck15] D.J. Eck. *Introduction to Programming Using Java, Seventh Edition*. 2015. URL: `http://math.hws.edu/javanotes/`. Capítulo 5 (5.5, 5.6 y 5.7).

[GJo15] J. Gosling, B. Joy y otros. *The Java® Language Specification, Java SE 8 Edition*, 2015. URL: `https://docs.oracle.com/javase/specs/jls/se8/html/`.

[Ora15] Oracle. *The Java™ Tutorials*, 2015. URL: `http://download.oracle.com/javase/tutorial/`. Trail: Learning the Java Language. Lesson: Language Basics - Interfaces and Inheritance. Lesson: Packages.

[SM16] W.J. Savitch. *Absolute Java, Sixth Edition*. Pearson Education, 2016. Capítulos 7 y 8.

[Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulos 2, 3 y 4.