16. Herencia

Objetivos:
a) Definir el concepto de herencia entre clases
b) Interpretar el código fuente de una aplicación Java donde aparecen clases relacionadas mediante la herencia
c) Construir una aplicación Java sencilla, convenientemente especificada, que haga uso de la herencia entre clases

16.1. Definición de herencia

La herencia es una propiedad que permite la declaración de nuevas clases a partir de otras ya existentes. Esto proporciona una de las ventajas principales de la Programación Orientada a Objetos: la reutilización de código previamente desarrollado ya que permite a una clase más específica incorporar la estructura y comportamiento de una clase más general.

Cuando una clase B se construye a partir de otra A mediante la herencia, la clase B hereda todos los atributos, métodos y clases internas de la clase A. Además la clase B puede redefinir los componentes heredados y añadir atributos, métodos y clases internas específicas.

Para indicar que la clase B (clase *descendiente*, *derivada*, *hija* o *subclase*) hereda de la clase A (clase *ascendiente*, *heredada*, *padre*, *base* o *superclase*) se emplea la palabra reservada **`extends`** en la cabecera de la declaración de la clase descendiente. La sintaxis es la siguiente:

```
public class ClaseB extends ClaseA {
    // Declaracion de atributos y metodos especificos de ClaseB
    // y/o redeclaracion de componentes heredados
}
```

Por ejemplo, a partir de la clase `Precio`:

```
/**
 *  Ejemplo de declaracion de la clase Precio
 *  A. Garcia-Beltran - noviembre, 2005
 */
public class Precio {
    // Variable de instancia
    public double euros;
    // Metodos publicos
    public double da() {
        return euros;
    }
    public void pone(double x) {
        euros=x;
    }
}
```

se construye la clase `Producto` como descendiente de la clase `Precio` de la siguiente forma:

```
/**
 *  Ejemplo de declaracion de la clase Producto
 *  clase producto desciende de Precio
```

```
 *  A. Garcia-Beltran - diciembre, 2005
 */

public class Producto extends Precio {
    // Variable de instancia
    public int codigo;
    // Metodos publicos
    public int daCodigo() {
        return codigo;
    }
    public void asignaCodigo(int x) {
        codigo=x;
    }
    public void asignaProducto(int cod, double p) {
        asignaCodigo(cod);
        pone(p);
    }
    public String toString() {
        return "Codigo: " + codigo + " ; precio: " + euros + " euros";
    }
}
```

La clase `PruebaClaseProducto` trabaja con dos instancias de la clase `Producto`:

```
/**
 *  Demostracion de la clase Producto
 *  A. Garcia-Beltran - diciembre, 2005
 */
public class PruebaClaseProducto {
    public static void main (String [] args){
        Producto p = new Producto();
        p.asignaProducto(200201,15.8);
        System.out.println(p.toString());
        Producto q = new Producto();
        q.asignaCodigo(200202);
        q.pone(34.3);
        System.out.println(q.toString());
    }
}
```

Durante la ejecución del código anterior, se generan las instancias (Figura 16.1), referenciadas por `p` y `q`, cada una de las cuales está compuesta por dos atributos: `euros`, variable de instancia heredada de la clase `Precio` y `codigo`, variable de instancia específica de la clase `Producto`.

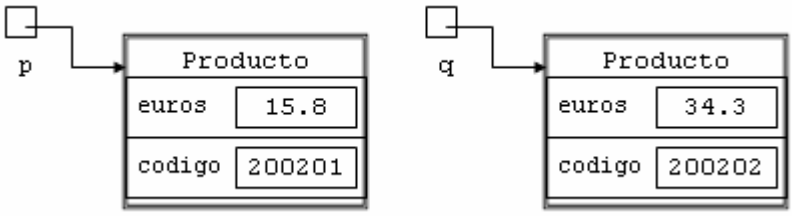

Figura 16.1. Representación grafica de las instancias de la clase `Producto`.

Por otro lado, la ejecución de `PruebaClaseProducto` produce la siguiente salida por pantalla:

```
$>javac PruebaClaseProducto.java↵

$>java PruebaClaseProducto↵

Codigo: 200201 ; precio: 15.8 euros
Codigo: 200202 ; precio: 34.3 euros
```

16.2. Jerarquía de clases

Java permite múltiples niveles de herencia pero no la herencia *multiple*, es decir una clase sólo puede heredar directamente de una clase ascendiente. Por otro lado, una clase puede ser ascendiente de tantas clases descendiente como se desee (*un unico padre, multitud de hijos*). En la Figura 16.2 se muestra gráficamente un ejemplo de jerarquía entre diferentes clases relacionadas mediante la herencia.

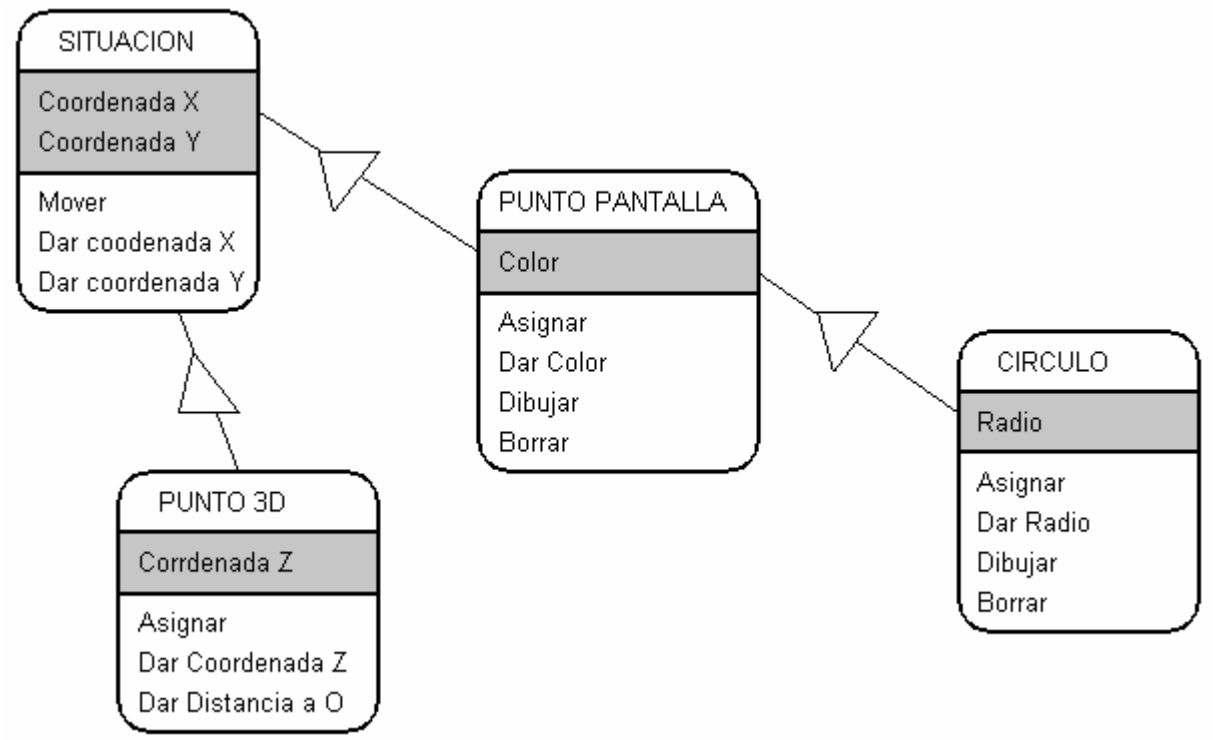

Figura 16.2. Representación de una jerarquía de clases relacionadas mediante la herencia

16.3. Redefinición de elementos heredados

Como se ha comentado anteriormente la clase descendiente puede añadir sus propios atributos y métodos pero también puede sustituir u ocultar los heredados. En concreto:

1. Se puede declarar un nuevo **atributo** con el mismo identificador que uno heredado, quedando este atributo **oculto**. Esta técnica no es recomendable.
2. Se puede declarar un nuevo **método de instancia** con la misma cabecera que el de la clase ascendiente, lo que supone su **sobreescritura**. Por lo tanto, la sobreescritura o redefinición consiste en que métodos adicionales declarados en la clase descendiente con el mismo nombre, tipo de dato devuelto y número y tipo de parámetros sustituyen a los heredados.

3. Se puede declarar un nuevo **método de clase** con la misma cabecera que el de la clase ascendiente, lo que hace que éste quede **oculto**. Por lo tanto, los métodos de clase o estáticos (declarados como `static`) no pueden ser redefinidos.
4. Un método declarado con el modificador `final` tampoco puede ser redefinido por una clase derivada.
5. Se puede declarar un **constructor** de la subclase que llame al de la superclase de forma implícita o de mediante la palabra reservada `super`.
6. En general puede accederse a los métodos de la clase ascendiente que han sido redefinidos empleando la palabra reservada `super` delante del identificador del método. Este mecanismo sólo permite acceder al metodo perteneciente a la clase en el nivel inmediatamente superior de la jerarquía de clases.

16.4. La clase Object

Independientemente de utilizar la palabra reservada `extends` en su declaración, todas las clases derivan de una superclase llamada `Object`. Ésta es la clase raíz de toda la jerarquía de clases de Java (Figura 16.3). El hecho de que todas las clases deriven implícitamente de la clase `Object` no se considera herencia múltiple.

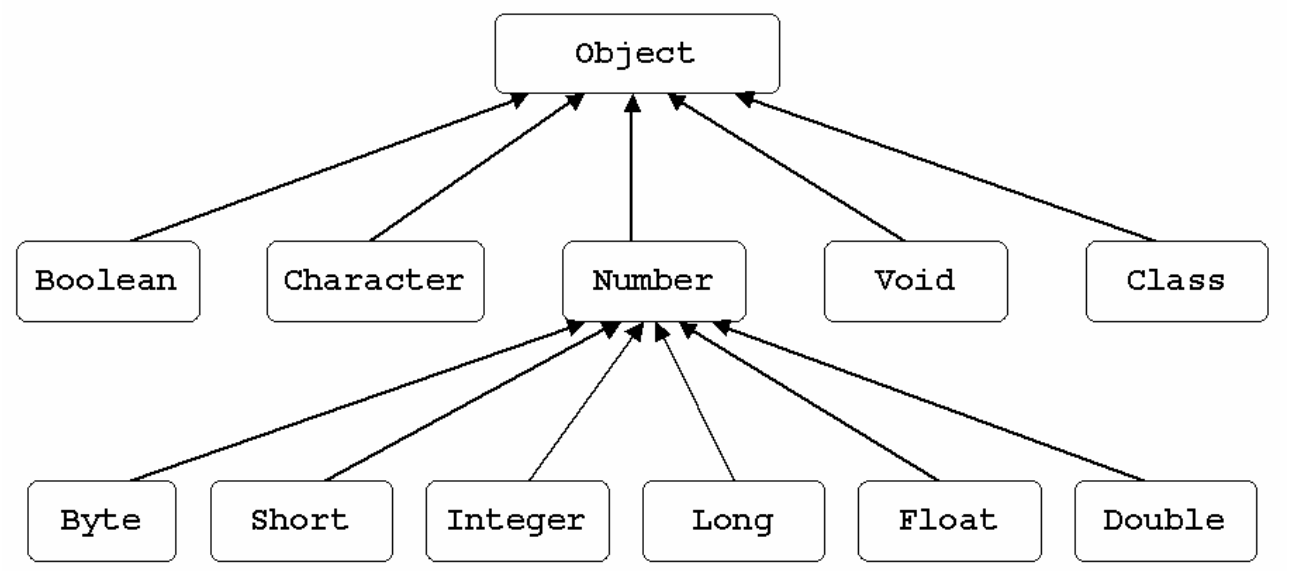

Figura 16.3. Jerarquía de clases predefinidas en Java

Como consecuencia de ello, todas las clases tienen algunos métodos heredados de la clase `Object` (Tabla 16.1).

Tabla 16.1. Algunos de los métodos de la clase predefinida `Object`

Método	Función
`clone()`	Genera una instancia a partir de otra de la misma clase.
`equals()`	Devuelve un valor lógico que indica si dos instancias de la misma clase son iguales.
`toString()`	Devuelve un `String` que contiene una representación como cadena de caracteres de una instancia.
`finalize()`	Finaliza una instancia durante el proceso de recogida de basura (se verá más adelante).

`hashCode()`	Devuelve una clave *hash* (su dirección de memoria) para la instancia
`getClass()`	Devuelve la clase a la que pertenece una instancia.

Es bastante frecuente tener que sobreescribir algunos de estos métodos. Por ejemplo, para verificar si dos instancias son iguales en el sentido de contener la misma información en sus atributos se debería sobreescribir el método `equals()`. El siguiente código muestra un ejemplo de módificación de la clase `Producto` para incluir una sobreescritura del método `equals()`:

```
public class Producto extends Precio {
    ...
    public boolean equals(Object a) {
        if (a instanceof Producto)
            return (codigo==a.daCodigo());
        else
            return false;
    }
}
```

También es bastante habitual sobreescribir el método `toString()`.

16.5. Herencia y constructores

La subclase necesita normalmente que se ejecute el constructor de la superclase antes que su propio constructor para inicializar las variables de instancia heredadas. La solución consiste en utilizar la palabra reservada `super` seguida entre paréntesis de los parámetros correspondiente en el cuerpo del constructor de la subclase. Es decir, incluir la siguiente sentencia como primera línea de código:

```
super(argumentos opcionales);
```

De esta forma la implementación de un constructor de la clase descendiente sólo necesita inicializar directamente las variables de instancia no heredadas. Si no aparece como primera sentencia, el compilador inserta una llamada implícita `super();` que inicializa las variables de instancia a cero, `false`, carácter nulo o `null` dependiendo de su tipo. Esta llamada en cadena a los constructores de las clases ascendientes llega hasta el origen de la jerarquía de clases, es decir, hasta el constructor de la clase `Object`. En cualquier caso, la creación de una nueva instancia mediante un constructor debe tener tres fases:

1. Llamada al constructor de la clase ascendiente
2. Se asignan valores a los atributos
3. Se ejecuta el resto del constructor

16.6. Casting o moldes entre objetos con relación de herencia

El *casting* o moldeo permite el uso de un objeto de una clase en lugar de otro de otras clase con el que haya una relación de herencia. Por ejemplo:

```
Object a = new Producto();
```

Entonces `a` es *momentáneamente* tanto una instancia de la clase `Object` como `Producto` (hasta que más adelante se le asigne un objeto que no sea un `Producto`). A esto se le llama moldeo **implícito**.

Por otro lado, si se escribe:

```
Producto b = a;
```

se obtendrá un error de compilación porque el objeto referenciado por `a` no es considerado por el compilador como un `Producto`. Sin embargo se le puede indicar al compilador que a la referencia `a` se le va a asignar obligatoriamente un `Producto`.

```
Producto b = (Producto)a;
```

Este moldeo **explícito** introduce la verificación durante la ejecución de que a la referencia `a` se le ha asignado un `Producto` así que el compilador no genera un error. En el caso que durante la ejecución la referencia `a` no fuera a un `Producto`, se generaría una excepción. Para asegurar esta situación y evitar el error de ejecución se podría emplear el operador `instanceof`:

```
if (a instanceof Producto) {
   Producto b = (Producto)a;
}
```

16.7. Clases y métodos abstractos

Una clase *abstracta* es una clase de la que no se pueden crear instancias. Su utilidad consiste en permitir que otras clases deriven de ella. De esta forma, proporciona un modelo de referencia a seguir a la vez que una serie de métodos de utilidad general. Las clases abstractas se declaran empleando la palabra reservada `abstract` como se muestra a continuación:

```
public abstract class IdClase . . .
```

Una clase abstracta puede componerse de varios atributos y métodos pero debe tener, al menos, un método *abstracto* (declarado también con la palabra reservada `abstract` en la cabecera). Los métodos abstractos no se implementan en el código de la clase abstracta pero las clases descendientes de ésta han de implementarlos o volver a declararlos como abstractos (en cuyo caso la subclase también debe declararse como abstracta). En cualquier caso, ha de indicarse el tipo de dato que devuelve y el número y tipo de parámetros. La sintaxis de declaración de un método abstracto es:

```
abstract modificador tipo_retorno idClase(lista_parametros);
```

Si una clase tiene métodos abstractos, entonces también la clase debe declararse como abstracta. Como los métodos de clase (`static`) no pueden ser redefinidos, un método abstracto no puede ser estático. Tampoco tiene sentido que declarar constructores abstractos ya que un constructor se emplea siempre al crear una instancia (y con las clases abstractas no se crean instancias).

Ejemplo de código con la declaración de clase abstracta:

```
/**
 *  Declaracion de la clase abstracta FiguraGeometrica
 *  A. Garcia-Beltran - noviembre, 2005
 */
public abstract class FiguraGeometrica {
    // Declaracion de atributos
    private String nombre;
    // Declaracion de metodos
    abstract public double area();
    public figuraGeometrica (String nombreFigura ) {
        nombre = nombreFigura;
    }
    final public boolean mayorQue (FiguraGeometrica otra) {
        return area()>otra.area();
    }
    final public String toString() {
        return nombre + " con area " + area();
    }
}
```

Como ejemplo de utilización de una clase abstracta en el siguiente código la clase `Rectangulo` se construye a partir de la clase abstracta `FiguraGeometrica`:

```
/**
 *  Ejemplo de uso de la declaracion de una clase abstracta
 *  Declaracion de la clase Rectangulo
 *  A. Garcia-Beltran - noviembre, 2005
 */
public class Rectangulo extends FiguraGeometrica {
    private double base;
    private double altura;
    public Rectangulo (double largo, double ancho) {
        super("Rectangulo");
        base=largo;
        altura=ancho;
    }
    public double area () {
        return base * altura;
    }
}
```

Ejemplo de uso de la clase `Rectangulo`:

```
/**
 *  Ejemplo de uso de la clase Rectangulo
 *  A. Garcia-Beltran - noviembre, 2005
 */
public class pruebaRectangulo {
    public static void main (String [] args ) {
        Rectangulo r1;
        r1 = new Rectangulo(12.5, 23.7);
        System.out.println("Area de r1 = " + r1.area());
        Rectangulo r2 = new Rectangulo(8.6, 33.1);
        System.out.println("Area de r2 = " + r2.toString());
        if (r1.mayorQue(r2))
            System.out.println("El rectangulo de mayor area es r1");
            else System.out.println("El rectangulo de mayor area es r2");
    }
}
```

Salida por pantalla de la ejecución del código anterior:

```
$>java PruebaRectangulo↵

Area de r1 = 296.25
Area de r2 = Rectangulo con area 284.66
El rectangulo de mayor area es r1
```

16.8. Clases y métodos finales

Una clase declarada con la palabra reservada `final` no puede tener clases descendientes. Por ejemplo, la clase predefinida de Java `Math` está declarada como `final`.

A modo de ejemplo, se desarrolla una clase **final** `MathBis` (de operatividad similar a la clase `Math` estándar de Java) que incluye la declaración de dos métodos que calculan y devuelven respectivamente las siguientes funciones trigonométricas:

$$\text{senh}^{-1}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$$

$$\cosh^{-1}(x) = \ln\left(x + \sqrt{x^2 - 1}\right)$$

El código fuente de la clase es:

```
/**
 *  Ejemplo de declaracion de una clase final
 *  Declaracion de la clase MathBis
 *  A. Garcia-Beltran - diciembre, 2004
 */
public final class MathBis {
   public static double asinh(double x) {
      return Math.log(x+Math.sqrt(x*x+1));
   }
   public static double acosh(double x) {
      return Math.log(x+Math.sqrt(x*x-1));
   }
}
```

Ejemplo de uso de la clase `MathBis`:

```
/**
 *  Ejemplo de uso de una clase final
 *  Declaracion de la clase pruebaMathBis
 *  A. Garcia-Beltran - diciembre, 2004
 */
public class PruebaMathBis {
   public static void main (String [] args) {
      for (int i=-5; i<10; i++) {
          double x = i/5.0;
          System.out.print("Para x = " + x);
          System.out.print(": asinh(x) = " +MathBis.asinh(x));
          System.out.println(", acosh(x) = " +MathBis.acosh(x));
      }
   }
}
```

Salida por pantalla de la ejecución del código anterior:

```
$>java PruebaMathBis↵

Para x = -1.0: asinh(x) = -0.8813735870195428, acosh(x) = NaN
Para x = -0.8: asinh(x) = -0.7326682560454109, acosh(x) = NaN
Para x = -0.6: asinh(x) = -0.5688248987322477, acosh(x) = NaN
Para x = -0.4: asinh(x) = -0.39003531977071504, acosh(x) = NaN
Para x = -0.2: asinh(x) = -0.19869011034924128, acosh(x) = NaN
Para x = 0.0: asinh(x) = 0.0, acosh(x) = NaN
Para x = 0.2: asinh(x) = 0.19869011034924142, acosh(x) = NaN
Para x = 0.4: asinh(x) = 0.3900353197707155, acosh(x) = NaN
Para x = 0.6: asinh(x) = 0.5688248987322475, acosh(x) = NaN
Para x = 0.8: asinh(x) = 0.732668256045411, acosh(x) = NaN
Para x = 1.0: asinh(x) = 0.8813735870195429, acosh(x) = 0.0
Para x = 1.2: asinh(x) = 1.015973134179692, acosh(x) = 0.6223625037147785
Para x = 1.4: asinh(x) = 1.1379820462933672, acosh(x) = 0.867014726490565
Para x = 1.6: asinh(x) = 1.2489833279048763, acosh(x) = 1.0469679150031885
Para x = 1.8: asinh(x) = 1.3504407402749723, acosh(x) = 1.1929107309930491
```

Por otro lado, un método declarado como `final` no puede ser redefinido por una clase descendiente. Los métodos que son llamados desde los constructores deberían declararse como `final`, ya que si un constructor llama a un método que no lo sea, la subclase podría haberla redefinido con resultados indeseables.