

Resumen PAV (2)

2006-mar-27

Este resumen ha sido elaborado para ser leído después de haber asistido a clase: pretende fijar y profundizar en los conceptos ya explicados.

Herencia (*inheritance*)

Introducción

Una de las características de Java y de otros lenguajes orientados a objetos es la posibilidad de “reusar el código”: reusar en nuestros programas clases que ya han sido escritas anteriormente (por nosotros mismos o por otros programadores), sin tener que reinventar la rueda.

En la discusión que sigue, viene bien recordar que los objetos tienen un *estado* (que se refleja en sus variables o atributos) y un *comportamiento* (que viene dado por sus métodos). Cuando usemos clases que han escrito otras personas, buscaremos clases que implementen el comportamiento y reflejen el estado que necesitamos en nuestro programa.

En líneas generales hay **dos modos** de reusar clases:

a) por composición (*composition*)

Es lo que hemos hecho repetidas veces en el curso: nuestra clase usa otras clases como atributos.

Por ejemplo:

```
public class Game {  
    Room currentRoom;  
    (...)  
}
```

Este modo se llama composición porque creamos objetos de otras clases dentro de nuestra clase (en el ejemplo, Room dentro de Game) y reusamos su funcionalidad. Los métodos de Room no son métodos de nuestra clase. Siguen siendo los métodos de Room, pues para llamarlos necesitamos hacerlo a través de `currentRoom`. Por ejemplo,

```
System.out.println( currentRoom.longDescription() );
```

b) por herencia (*inheritance*)

El segundo modo es más sutil. Se trata de crear una nueva clase que es *del tipo* de una clase existente (podríamos decir que creamos la nueva clase usando como base la plantilla de la clase original), pero añadiendo además las variables y métodos que necesitamos *sin modificar el código de la clase original*. Esta técnica se llama *herencia (inheritance)*, y es una de las piedras sillares de la programación orientada a objetos.

Llamamos **superclase** a la clase base, y llamamos **subclase** a la clase que hereda los métodos (el comportamiento) y atributos (el estado) de la superclase. También se habla de clase madre y clase hija. Para relacionar la superclase y la subclase (en otras palabras, para indicar al compilador que una

clase hereda de otra) se usa la palabra reservada **extends**.

Por ejemplo, supongamos que queremos crear un nuevo tipo de cuarto, que tiene la funcionalidad de la clase Room original. Escribiríamos:

```
public class RoomConItems extends Room {  
}
```

```
graph LR
    subclase[Subclase] --> RoomConItems
    Superclase[Superclase] --> Room
```

Con las pocas líneas que hemos escrito arriba, sin añadir nada más, tenemos una nueva clase RoomConItems que contiene todos los métodos y atributos de la clase original. Como no hemos añadido todavía ningún código a la clase extendida, el comportamiento de ambas clases es exactamente igual. Aunque no hayamos escrito una sola línea de código, ya podemos decir:

```
RoomConItems cuarto = new RoomConItems("fotocopiadora");  
System.out.println( cuarto.longDescription() );
```

Para usar como superclase una clase determinada, no necesitamos conocer su código fuente. Basta tener el código compilado (como sucede las clases que forman parte de la librería estándar de Java: ArrayList, File, InputStreamReader... no tenemos la fuente de ninguna de las clases, pero podemos usarlas).

Especialización de clases

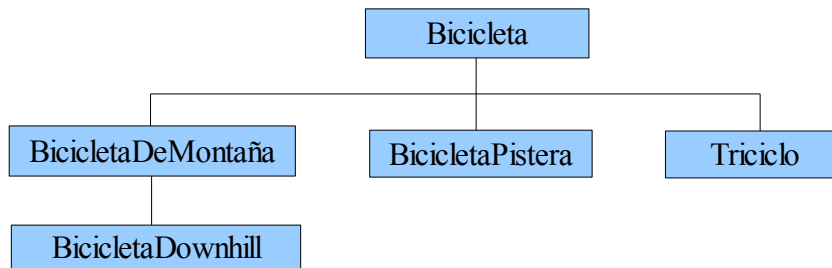
Cuando diseñamos una nueva clase heredando de otra, estamos *especializando* el comportamiento de la clase base. Éste es un buen punto de vista a tener en cuenta cuando diseñamos nuestros programas, y nos ayuda a discernir cuándo debemos usar composición y cuándo herencia. En el ejemplo de arriba, podríamos pensar que RoomConItems es una clase Room que implementará la posibilidad de que el cuarto contenga items (objetos: papel, tijera, puñal, veneno...)

El típico ejemplo del típico libro de Java considera una clase Bicicleta:

a) Usando *composición*, la clase base define los elementos comunes a todas las bicicletas (usando otras clases que se supone se han definido en otro momento):

```
public class Bicicleta {  
    Timon        timon;    // timón de la bicicleta, usando la clase Timon  
    Rueda        ruedaDelantera;  
    Rueda        ruedaTrasera;  
    Catalina[]   catalinas;  
    Velocidades[] velocidades;  
    double       peso;  
  
    /* (...) otros atributos y métodos de la clase base */  
}
```

Nuestro “árbol” de clases Bicicleta podría ser algo así:



BicicletaDeMontana, BicicletaPistera y Triciclo son subclases directas de Bicicleta. BicicletaDownhill es un subclase directa de BicicletaDeMontana. Indirectamente también es una subclase de Bicicleta, pues hereda de Bicicleta a través de BicicletaDeMontana.

Parte del código que podríamos usar para definir estas subclases:

```
public class BicicletaDeMontana extends Bicicleta {
    (...)
}

public class BicicletaDownhill extends BicicletaDeMontana {
    (...)
}
```

Algunas afirmaciones importantes que podemos hacer a partir del árbol genealógico de Bicicleta:

BicicletaDeMontana, BicicletaPistera, Triciclo, BicicletaDownHill son objetos de tipo Bicicleta (Bicicleta es un objeto más general que sus clases especializadas). Por eso, como BicicletaDeMontana tiene todos los atributos y métodos de Bicicleta, se puede decir:

```
BicicletaDeMontana bmx = new BicicletaDeMontana();
Bicicleta cletaSencilla = bmx;
```

En cambio, Bicicleta no tiene todos los atributos y métodos de BicicletaDeMontana, y por tanto no puedo decir:

```
Bicicleta cletaSencilla = new Bicicleta();
BicicletaDeMontana bmx = cletaSencilla;
```

Cómo afecta la herencia a las subclases

Constructores

Como ya se dijo, las subclases heredan todos los métodos y atributos de la superclase. Vamos a definir una regla:

Los constructores de las subclases deben llamar a alguno de los constructores de la superclase.

De lo contrario, Java llamará automáticamente al constructor por defecto de la superclase.

Pongamos un ejemplo para entender mejor la regla. Consideremos una clase que ha definido un constructor:

```
public class ClaseBase {
    private String nombre;

    /**
     * Constructor
     */
    public ClaseBase(String nombre) {
        this.nombre = nombre;
    }

    public String getNombre() {
        return nombre;
    }
}
```

Ahora definimos la subclase:

```
public class SubClase extends ClaseBase {

}
```

¿Qué sucede si tratamos de compilar SubClase?¹ El compilador nos da el siguiente error: **cannot find symbol: Constructor ClaseBase()**². Tratemos de entender lo que está pasando:

a) Cuando no definimos el constructor de una clase, Java suple con un constructor por defecto, que no tiene parámetros. En el caso de ClaseBase, este constructor por defecto sería ClaseBase(). Pero ClaseBase sí define un constructor: public ClaseBase(String nombre). Por tanto, Java no inserta el constructor ClaseBase().

b) SubClase no tiene un constructor, de modo que Java suple creando detrás de las bambalinas un constructor por defecto: SubClase().

c) Hemos dicho que los constructores de la subclase deben llamar a alguno de los constructores de la superclase. En el caso de SubClase, el constructor SubClase() insertado por Java intenta llamar al constructor por defecto de su superclase: ClaseBase(). Pero nosotros hemos definido un constructor ClaseBase(String nombre) y, en cambio, el constructor ClaseBase() no existe. Por tanto, el compilador se queja diciendo que no encuentra el constructor ClaseBase().

Para solucionar el *impasse*, tenemos que definir un constructor en SubClase que llame a un constructor que ya esté definido en ClaseBase³.

¹ Cree un nuevo proyecto en BlueJ, cree las dos clases e intente compilarlo.

² No puedo encontrar el símbolo: Constructor ClaseBase.

³ También podríamos definir un constructor ClaseBase() en la clase ClaseBase, pero la idea no es retocar la superclase, sino usarla como si no conociéramos el código fuente.

```

public class SubClase extends ClaseBase {

    /**
     * Constructor de SubClase
     */
    public SubClase(String s) {
        super(s); // llamamos al constructor
                  // de ClaseBase
    }
}

```

Ahora sí, el código compila sin errores. La palabra reservada `super` se usa para hacer referencia a la superclase⁴.

Es importante caer en la cuenta de por qué Java nos fuerza a llamar al constructor de la superclase. La superclase tiene sus propios constructores y métodos que aseguran que la clase se cree correctamente y funcione correctamente. Cuando creamos un objeto de la subclase usando `new` (`SubClase o = new SubClase();`) **estamos creando implícitamente, antes que nada, un objeto de la superclase**. Por tanto, es necesario que se ejecute el constructor de la superclase para que “esa parte” de nuestro nuevo objeto se cree correctamente. Y por eso Java nos obliga a llamar al constructor de la superclase.

Definamos otra regla: *todos los métodos y atributos de la superclase son parte de la subclase. Sin embargo, la subclase sólo puede acceder aquellos métodos y atributos que no hayan sido declarados private.*

En nuestro ejemplo, eso quiere decir que:

- `String nombre` de `ClaseBase` es `private`. Por tanto, aunque está en la subclase, la subclase no lo puede acceder. En otras palabras, dentro de `SubClase` no puedo decir `System.out.println(nombre)`⁵; Si quiero acceder a `nombre`, tengo que decir `System.out.println(getNombre())`;
- No necesito decir `super.getNombre()`, pues el método `getNombre` está definido como `public` en `ClaseBase`. Por tanto, es un método público de `SubClase`.

En nuestro ejemplo:

```

public class SubClase extends ClaseBase {

    /**
     * Constructor de SubClase
     */
    public SubClase(String s) {
        super(s);
    }

    public void imprimeNombre() {
        System.out.println( getNombre() );
    }
}

```

Una tercera clase puede usar `getNombre()` de `SubClase` como si `SubClase` hubiese definido el

⁴ Podemos imaginar que decir `super(nombre)` es como decir `ClaseBase(nombre)`.

⁵ El compilador se quejará diciendo: **nombre has private access in ClaseBase.**

método:

```
class OtraClass {
    public static void main(String[] args) {
        SubClase sc = new SubClase("Animal");
        System.out.println( sc.getNombre() );
    }
}
```

Reemplazando (*overriding*) y sobrecargando (*overloading*) los métodos de la superclase

Las subclases pueden reemplazar los métodos de su superclase. Este proceso recibe el nombre de *override*. Podemos dar una regla al respecto:

Cuando una subclase define un método que tiene la misma firma⁶ (signature) y tipo de retorno que un método de la superclase, este nuevo método reemplaza al método de la superclase.

Nótese que no basta que los métodos tengan la misma firma: deben devolver el mismo tipo de datos⁷. Si intento redefinir una clase con la misma firma, el compilador me dirá que no es posible, pues el método ya ha sido definido.

El método original de la superclase puede seguirse usando mediante la palabra reservada `super`. En el siguiente código redefinimos el método `getNombre()` de la subclase, y dentro del nuevo método `getNombre()` usamos `super.getNombre()` para acceder al nombre:

```
public class SubClase extends ClaseBase
{
    public SubClase(String nombre) {
        super(nombre);
    }

    public static void main(String[] args) {
        SubClase o = new SubClase("Animal");
        System.out.println(o.getNombre());
    }

    public String getNombre() {
        String s = super.getNombre();
        return "El nombre es " + s;
    }
}
```

6 Como sabemos de memoria, la firma o signature de un método se compone del nombre del método y de sus argumentos: `lanzaMisil(String nombre)` tiene una firma distinta que la de `lanzaMisil(int numMisil)`, pues aunque los nombres de los métodos son iguales, los parámetros son de distinto tipo (el nombre que demos a la variable del argumento no interesa para nada). Como las firmas son distintas, son dos métodos distintos.

7 También se puede definir un método que devuelva una subclase del método que devolvía la clase original. Esta posibilidad se introdujo en Java5, y se llama *covariant return type*. Sin embargo, a efectos del curso, no nos interesa complicarnos con este detallito: imaginemos que este pie de página nunca existió.

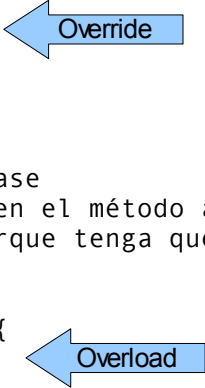
Todas las clases que creen una instancia de la clase SubClase y llamen al método getNombre() estarán llamando al método de SubClase, y no al de ClaseBase.

Cuando se define un método que tiene el mismo nombre que la superclase, pero tiene una firma distinta (tiene distintos argumentos), estamos definiendo un método distinto: los dos métodos (el de la superclase y el de la subclase) pueden convivir pacíficamente. Técnicamente se dice que el método está **sobrecargado** (*overloaded*).

Resumiendo: se dice *override* cuando el método de la superclase es reemplazado en la subclase; se dice *overload* cuando la subclase define un método del mismo nombre que uno método de la superclase, pero los métodos tienen parámetros distintos.

```
public class SubClase extends ClaseBase
{
    /* (...) igual que el ejemplo anterior */
    /**
     * Override del método de la superclase
     */
    public String getNombre() {
        String s = super.getNombre();
        return "El nombre es " + s;
    }

    /**
     * Overload del método de la superclase
     * (aquí uso super.getNombre porque en el método anterior
     * he reemplazado getNombre(), no porque tenga que ver
     * con el overloading)
     */
    public String getNombre(String msg) {
        return msg + super.getNombre();
    }
}
```



The diagram consists of two blue arrows pointing to the right. The first arrow, labeled 'Override', points to the `getNombre()` method in the code block. The second arrow, labeled 'Overload', points to the `getNombre(String msg)` method in the code block.

Override y overload de métodos y variables estáticos.

El lector/lectora astuto se estará preguntando qué pasa cuando una subclase declara un método con el mismo nombre y firma que un método estático⁸ de la superclase.

Enunciemos otra regla:

*Cuando una subclase declara un método estático con la misma firma que un método estático de su superclase, el método estático de la superclase **no es reemplazado**, sino que queda oculto.*

lo que quiere decir que persisten los dos métodos estáticos (el de la superclase y el de la subclase). Qué método se ejecuta depende de qué tipo de si es la superclase o la subclase la que llama al método.

⁸ Algunos libros llaman a los métodos estáticos *class methods*.

Consideremos la siguiente clase, con un método estático y un método normal.

```
public class Animal
{
    public static void metodoEstatico() {
        System.out.println("El método estático de Animal.");
    }

    public void noEstatico() {
        System.out.println("El método noEstatico() de animal.");
    }
}
```

Definamos una subclase, que declare nuevamente metodoEstatico() y noEstatico():

```
public class Gato extends Animal {

    public static void metodoEstatico() {
        System.out.println("El método estático de Gato.");
    }

    public void noEstatico() {
        System.out.println("El método noEstatico() de gato.");
    }

    public static void main(String[] args) {
        Gato gato = new Gato();
        Animal animal = gato; // podemos decir esto porque todo Gato
                               // es Animal. (Aunque no todo Animal es Gato).

        System.out.println("Método estático:");
        animal.metodoEstatico(); // no decimos Animal.metodoEstatico
        Gato.metodoEstatico();    // para que quede claro que el
                               // método que es el del objeto new Gato()

        System.out.println("Método no estático:");
        animal.noEstatico();
        gato.noEstatico();
    }
}
```

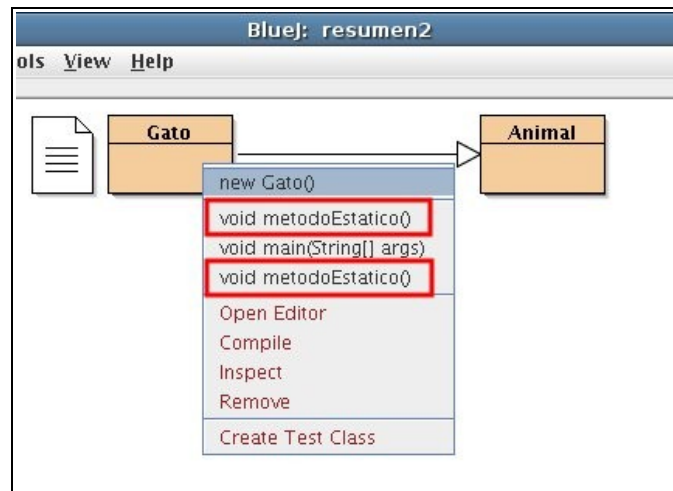
Si ejecutamos main de Gato, la salida es la siguiente:

```
Método estático:
El método estático de Animal.
El método estático de Gato.
Método no estático:
El método no estático de Gato.
El método no estático de Gato.
```

Lo que demuestra que si intentamos reemplazar en la subclase un método estático de la superclase, en realidad **el método no es reemplazado sino que persisten los dos métodos**⁹. Para que quede más claro todavía, BlueJ nos muestra que la clase Gato tiene dos métodos estáticos del mismo

⁹ Si quisiéramos llamar al método noEstatico() de Animal, tendríamos que usar super.noEstatico() dentro de Gato.

nombre:



Lo que sucede cuando se declara en la subclase una variable estática con el mismo nombre que una variable estática de la superclase, el efecto es el mismo que con los métodos: las dos variables persisten, y cuál se usa en cada caso depende de qué clase es el objeto que contiene la instancia de la subclase.

Podemos enunciar otra regla:

*Cuando una subclase declara una variable estática con del mismo tipo y nombre una variable estática de su superclase, la variable estática de la superclase **no es reemplazada**, sino que queda oculta.*

Modifiquemos nuestras clases `Animal` y `Gato`, añadiendo una variable estática `test`, y un constructor para cada clase que inicialice la variable con valores distintos según se trate de `Animal` o `Gato`:

```
public class Animal
{
    static String test;

    public Animal() {
        test = "Animal";
    }
    (... el resto igual)
}
```

```
public class Gato extends Animal {

    static String test;

    public Gato() {
        super(); // tenemos que llamar al constructor de Animal
    }
}
```

```

        test = "Gato";
    }

    (...)
    public static void main(String[] args) {
        (...)
        Gato.metodoEstatico();

        // Aquí probamos si la variable es reemplazada o simplemente
        // persisten las dos
        System.out.println("\ntest en Animal: " + animal.test);
        System.out.println("test en Gato: " + gato.test + "\n");

        (...)
    }
}

```

La salida del programa es ahora:

```

Método estático:
El método estático de Animal.
El método estático de Gato.

test en Animal: Animal
test en Gato: Gato

Método no estático:
El método no estático de Gato.
El método no estático de Gato.

```

lo que muestra que persisten las dos variables estáticas. Se puede hacer la prueba con dos variables no estáticas, para comprobar que efectivamente la de la superclase es reemplazada.

Como se puede apreciar, la redeclaración de variables o métodos estáticos de la superclase en la subclase se presta a confusión. Por esto, ésta técnica, que se denomina *ocultar las variables*, está vivamente desaconsejada¹⁰.

Control de acceso

a) Métodos y variables `private`

Cuando una superclase declara un método o variable como `private`, la subclase hereda el método o variable. Sin embargo, no puede acceder al método ni a la variable (tampoco usando `super`).

Si queremos definir un método o variable de modo tal que sea `public` para las subclases pero `private` para todas las demás clases, declaramos el método de la superclase como `protected`. De este modo la subclase puede manipular con comodidad las variables de la superclase.

b) Las subclases, usando *override*, dar más acceso al método reemplazado que el acceso que tenía en la superclase, pero no pueden restringirlo. Por ejemplo, un método `protected` de la superclase puede declararse `public` en la subclase, pero no `private`.

10 cfr. Dante Alighieri, *La Divina Comedia*, Infierno, Canto XVII.

La clase Object

La clase `Object`, definida en el paquete `java.lang`¹¹, define e implementa el comportamiento que es común a todas las clases que existen y existirán en Java. Es la clase más general, y está en la cima de la jerarquía de clases. Todas las demás clases (las de la librería de Java y las que nosotros podamos crear) son subclases de esta clase primigenia, directa o indirectamente¹².

Algunos lenguajes permiten crear clases que heredan de más de una clase simultáneamente. Java, en cambio, sólo permite heredar de una clase a la vez. Por tanto, toda clase en Java tiene exactamente una superclase, salvo la clase `Object`, que es la madre de todas las clases.

La clase `Object` tiene algunos métodos interesantes que comentamos a continuación. Como todas las clases heredan de `Object`, todas las clases heredan también estos métodos y el programador los puede usar en su provecho.

El método `toString()`

El método `toString()` devuelve una representación de nuestro objeto en un objeto de tipo `String`. El uso típico de `toString()` es imprimir el estado de un objeto (sus variables) utilizando `System.out.println`:

```
System.out.println(libro.toString());
```

o, más abreviadamente,

```
System.out.println(libro)
```

El motivo por el cuál estos dos ejemplos dan el mismo resultado es que en la clase `System.out` el método `println` está sobrecargado¹³. Probablemente sea algo así:

```
public static void println(String s) {  
    // (...) aquí algo que imprime un string  
}  
  
public static void println(Object o) {  
    System.out.println(o.toString());  
}
```

De modo que cuando decimos `System.out.println(libro)`, como `libro` es finalmente de tipo `Object`, se ejecuta `println(Object o)` y no `println(String s)`.

Es una buena práctica de hacer un *override* del método `toString()` en las clases que definamos, aunque no siempre sea necesario.

11 Este paquete se incluye por defecto en todos los programas Java, no necesitamos importarlo explícitamente.

12 Directamente cuando una subclase dice `class Test extends Object { ... }` e indirectamente cuando no dice nada (`class Test { ... }`). No hay escapatoria, todos descienden de `Object`.

13 Hay distintos métodos `println` definidos para distintos tipos de argumentos.

El método equals()

El método `equals` compara dos objetos y devuelve `true` si son iguales y `false` si no lo son. La pregunta aquí es qué quiere decir que dos objetos son iguales...

Java permite compara objetos usando el operador `==`. Sin embargo éste operador tiene un comportamiento distinto según compare tipos primitivos¹⁴ u objetos. Cuando se comparan tipos primitivos, el operador `==` compara sus valores. En cambio, cuando se comparan objetos, el operador `==` devuelve `true` o `false` según se trate del mismo objeto o no, independientemente del valor de los atributos de los objetos.

Por ejemplo:

```
int i = 5; int j = 5;
System.out.println( i == j );
```

devuelve `true`.

En cambio:

```
Integer i = new Integer(5);
Integer j = new Integer(5);
System.out.println( i == j );
```

devuelve `false`, pues `i` y `j` son dos objetos y, por tanto, el operador `==` compara sus referencias en memoria, es decir, compara si son “físicamente” el mismo objeto o no.

```
Integer i = new Integer(5);
Integer j = i;
System.out.println( i == j ); // devuelve true

// pero esto imprime 6 en vez de 5, porque i y j son el mismo objeto:
j = 6;
System.out.println( i );
```

Por eso, la clase `Object` tiene un método `equals`, que está pensado para que el programador, usando *override*, pueda definir qué quiere decir exactamente que dos objetos de su clase son iguales.

Pensemos en la clase `Complejo`. Dos números complejos son iguales si sus partes reales e imaginarias son iguales. Una clase que maneja complejos podría definir `equals` así:

```
public class Complejo {
    double real;
    double imag;

    public boolean equals(Complejo c) {
        if ( (real == c.real) & (imag == c.imag) )
            return true;
        else
            return false;
    }
    (... otros métodos de la clase)
```

¹⁴ Los tipos primitivos son `boolean`, `char`, `byte`, `short`, `int`, `long`, `double`, `float`, `boolean...` y por supuesto `void`. Estos son los únicos tipos de datos en Java que no son objetos, aunque tienen su equivalente en la jerarquía de clases: `Integer`, por ejemplo.

```
}
```

Si `c1` y `c2` son dos objetos de tipo `Complejo`, entonces podríamos escribir:

```
if ( c1.equals(c2) ) {  
    // hacer algo  
}
```

En resumen, si necesitamos comparar dos objetos de alguna clase que hayamos definido, y el operador `==` no nos sirve (pues compara si los dos objetos son realmente el mismo, y no si los valores de los objetos son iguales) entonces necesitaremos escribir nuestro propio método `equals`.

Clases y métodos abstractos

Una clase abstracta es aquella que define sólo parcialmente la implementación de sus métodos.

Una clase abstracta contiene uno o más métodos que se han declarado (se define el control de acceso, el tipo de dato que devuelven, y sus argumentos), pero no se ha definido la implementación (el código que se ejecutará cuando se llame al método) y, por tanto, nunca se podrá crear un objeto de esta clase. Para poder usar la clase, es necesario derivar una subclase que complete la implementación de los métodos abstractos (lo que quiere decir que hay que escribir el código de las clases abstractas).

Para definir una clase abstracta:

- a) se usa la palabra reservada `abstract` en la definición de la clase;
- b) opcionalmente, uno o más de sus métodos pueden ser declarados abstractos;

Basta que una clase tenga un solo método abstracto para que toda la clase sea abstracta y, por tanto, sea necesario usar la palabra `abstract` también en la declaración de la clase (y no sólo en la declaración del método).

Por ejemplo:

```
abstract public class Figura  
{  
    protected int x, y; // las coordenadas de nuestra figura  
    abstract void draw(); // el método que dibuja la figura  
  
    public Figura(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}
```

La clase `Figura` no sirve para crear directamente un objeto. Su finalidad es definir el comportamiento base de todas las clases de tipo `Figura`. Dicho de otro modo, la clase abstracta define completamente la interfaz que usarán las clases derivadas.

En el mundo real, sucede igual: una figura no existe en cuanto tal, es un concepto universal. En cambio, existen figuras concretas: un cuadrado, un círculo, etc.

Podemos definir una clase `Cuadrado` que extiende a `Figura`:

```

public class Cuadrado extends Figura
{
    private int lado;

    public Cuadrado(int x, int y, int lado) {
        super(x, y); // llamamos al constructor de Figura
        this.lado = lado;
    }

    // Supongamos que existe un método Line(x0,y0, x1,y1) que
    // dibuja la línea definida por (x0,y0) y (x1,y1).
    public void draw()15 {
        Line(x, y, x+lado, y);
        Line(x+lado, y, x+lado, y+lado);
        Line(x+lado, y+lado, x, y+lado);
        Line(x, y+lado, x, y);
    }
}

```

Gracias a que todos los objetos que tienen a `Figura` como superclase son finalmente un objeto de tipo `Figura`, se puede escribir código como el siguiente:

Supongamos que tenemos un array `Figura[] shapes = new Figura[100]`; de objetos que se han ido definiendo a lo largo del programa. Estos objetos son de diversos tipos, pero descienden todos de `Figura` (`Círculo`, `Cuadrado`, etc.). Queremos dibujarlos en la pantalla. Podemos decir:

```

for(int i=0; i < shapes.length; i++) {
    shapes[i].draw();
}

```

En cada iteración, `shapes[i]` tendrá un objeto diferente. No sabemos de qué tipo exactamente, sólo que es un tipo que tiene a `Figura` como superclase. Por tanto, ése objeto tendrá un método `draw()` y podemos llamarlo para que se dibuje a sí mismo en pantalla.

Cómo evitar que se creen subclases o que se reemplazen nuestros métodos

En ocasiones es conveniente evitar que se deriven subclases de una clase determinada, o que se haga un *override* de alguno de los métodos de nuestra clase. La palabra reservada `final` exactamente eso.

Una subclase no puede reemplazar los métodos que han sido declarados final en su superclase.

Cuando se declara una clase `final`, no se pueden definir subclases a partir de ella¹⁶.

¹⁵ Nótese que usamos `x` e `y` directamente porque están declarados como `protected` en `Figura` (y, por tanto, las subclases de `Figura` pueden usar las dos variables como si fueran públicas).

¹⁶ Por ejemplo, la clase `String` de `java.lang` está declarada como `final` y, por tanto, no se pueden crear subclases a partir de ella.

Cuando se califica como `final` un método de una clase, aunque se puede crear una subclase y los métodos y atributos se hereden, el método declarado `final` no se puede reemplazar (no se puede *override*).

Un motivo para hacer un método `final` puede ser que tiene una implementación que no debe cambiar y es crítica para el funcionamiento de la clase y de sus subclases. Una regla práctica es que los métodos que se llaman desde un constructor normalmente deben declararse `final`. Si un constructor llama a un método que no está definido como `final`, una subclase podría redefinir éste método, y como resultado el comportamiento del constructor sería confuso (si no errado).

En resumen

- Todas las clases en Java, con excepción de la clase `Object`, tienen exactamente una superclase directa.
- Una clase hereda todas las variables y métodos de todas sus superclases, directas o indirectas (indirectas cuando su superclase es a la vez subclase de otra superclase).
- Una subclase puede reemplazar los métodos que hereda (*override*).
- Una subclase puede esconder las variables que hereda, y puede esconder los métodos estáticos que hereda. Esconder variables o métodos es considerado una pésima práctica de programación.
- La clase `Object` es la madre de todas las clases, está en la cima de la jerarquía de clases. Todas las clases son sus descendientes y heredan métodos de ella. Algunos métodos útiles de `Object` son `toString()` e `equals()`.
- Se puede evitar que se cree una subclase a partir de una clase declarando la clase como `final`. De modo similar, se puede evitar que un método de la superclase sea reemplazado en la subclase declarando el método `final`.
- No se pueden crear objetos a partir de clases abstractas. Una clase abstracta puede contener métodos abstractos (métodos que se han declarado pero no implementado). Para poder crear un objeto, es necesario crear una subclase que implemente todos los métodos abstractos.

Algunas explicaciones están tomadas de los tutorials de Java de la página de Sun Microsystems, del libro *Thinking in Java, 4th Edition*, de Bruce Eckel y de los apuntes del curso. Comentarios, correcciones y sugerencias: Roberto Zoia (roberto.zoia@gmail.com)

Salvo que alguien considere que hay partes que ya tienen un copyright más restrictivo: This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.