Herencia

Composición

En anteriores ejemplos se ha visto que una clase tiene datos miembro que son instancias de otras clases. Por ejemplo:

```
class Circulo {
        Punto centro;
        int radio;
        float superficie() {
                return 3.14 * radio * radio;
        }
}
```

Esta técnica en la que una clase se compone o contiene instancias de otras clases se denomina composición. Es una técnica muy habitual cuando se diseñan clases. En el ejemplo diríamos que un Circulo tiene un Punto (centro) y un radio.

Herencia

Pero además de esta técnica de composición es posible pensar en casos en los que una clase es una extensión de otra. Es decir una clase es como otra y además tiene algún tipo de característica propia que la distingue. Por ejemplo podríamos pensar en la clase Empleado y definirla como:

```
class Empleado {
    String nombre;
    int numEmpleado , sueldo;

    static private int contador = 0;
```

```
    Empleado(String nombre, int sueldo) {
        this.nombre = nombre;
        this.sueldo = sueldo;
        numEmpleado = ++contador;
    }

    public void aumentarSueldo(int porcentaje)
{
        sueldo += (int)(sueldo * aumento /
100);
    }

    public String toString() {
        return "Num. empleado " + numEmpleado
+ " Nombre: " + nombre +
                " Sueldo: " + sueldo;
    }
}
```

En el ejemplo el Empleado se caracteriza por un nombre (String) y por un número de empleado y sueldo (enteros). La clase define un constructor que asigna los valores de nombre y sueldo y calcula el número de empleado a partir de un contador (variable estática que siempre irá aumentando), y dos métodos, uno para calcular el nuevo sueldo cuando se produce un aumento de sueldo (método aumentarSueldo) y un segundo que devuelve una representación de los datos del empleado en un String.(método toString).

Con esta representación podemos pensar en otra clase que reuna todas las características de Empleado y añada alguna propia. Por ejemplo, la clase Ejecutivo. A los objetos de esta clase se les podría aplicar todos los datos y métodos de la clase Empleado y añadir algunos, como por ejemplo el hecho de que un Ejecutivo tiene un presupuesto.

Así diriamos que la clase Ejecutivo extiende o hereda la clase Empleado. Esto en Java se hace con la clausula **`extends`** que se incorpora en la definición de la clase, de la siguiente forma:

```
class Ejecutivo extends Empleado {
     int presupuesto;
     void asignarPresupuesto(int p) {
          presupuesto = p;
     }
}
```

Con esta definición un Ejecutivo es un Empleado que además tiene algún rasgo distintivo propio. El cuerpo de la clase Ejecutivo incorpora sólo los miembros que son específicos de esta clase, pero implícitamente tiene todo lo que tiene la clase Empleado.

A Empleado se le llama clase base o superclase y a Ejecutivo clase derivada o subclase.

Los objetos de las clases derivadas se crean igual que los de la clase base y pueden acceder tanto sus datos y métodos como a los de la clase base. Por ejemplo:

```
Ejecutivo jefe = new Ejecutivo( "Armando
Mucho", 1000);
jefe.asignarPresupuesto(1500);
jefe.aumentarSueldo(5);
```

Nota: La discusión acerca de los constructores la veremos un poco más adelante.

Atención!: Un Ejecutivo ES un Empleado, pero lo contrario no es cierto. Si escribimos:

```
Empleado curri = new Empleado ( "Esteban Comex
Plota" , 100) ;
curri.asignarPresupuesto(5000);   // error
```

se producirá un error de compilación pues en la clase Empleado no existe ningún método llamado asignarPresupuesto.

Redefinición de métodos. El uso de `super.`

Además se podría pensar en redefinir algunos métodos de la clase base pero haciendo que métodos con el mismo nombre y características se comporten de forma distinta. Por ejemplo podríamos pensar en rediseñar el método toString de la clase Empleado añadiendo las características propias de la clase Ejecutivo. Así se podría poner:

```
class Ejecutivo extends Empleado {
     int presupuesto;

     void asignarPresupuesto(int p) {
          presupuesto = p;
     }

     public String toString() {
          String s = super.toString();
          s = s + " Presupuesto: " +
presupuesto;
          return s;
     }
}
```

De esta forma cuando se invoque `jefe.toString()` se usará el método toString de la clase Ejecutivo en lugar del existente en la clase Empleado.

Observese en el ejemplo el uso de **`super`** , que representa referencia interna implícita a la clase base (superclase). Mediante **`super`**`.toString()` se invoca el método toString de la clase Empleado

Inicialización de clases derivadas

Cuando se crea un objeto de una clase derivada se crea implicitamente un objeto de la clase base que se inicializa con su constructor correspondiente. Si en la creación del objeto se usa el constructor no-args, entonces se produce una llamada implicita al constructor no-args para la clase base. Pero si se usan otros constructores es necesario invocarlos explicitamente.

En nuestro ejemplo dado que la clase método define un constructor, necesitaremos también un constructor para la clase Ejecutivo, que podemos completar así:

```
class Ejecutivo extends Empleado {
     int presupuesto;

     Ejecutivo (String n, int s) {
          super(n,s);
     }

     void asignarPresupuesto(int p) {
          presupuesto = p;
     }

     public String toString() {
          String s = super.toString();
          s = s + " Presupuesto: " +
presupuesto;
          return s;
     }
}
```

Observese que el constructor de Ejecutivo invoca directamente al constructor de Empleado mediante `super( argumentos )`. En caso de resultar necesaria la invocación al constructor de la superclase debe ser la primera sentencia del constructor de la subclase.

Herencia - II

El modificador de acceso `protected`

El modificador de acceso `protected` es una combinación de los accesos que proporcionan los modificadores `public` y `private`. `protected` proporciona acceso público para las clases derivadas y acceso privado (prohibido) para el resto de clases.

Por ejemplo, si en la clase Empleado definimos:

```
class Empleado {
     protected int sueldo;
     . . .
}
```

entonces desde la clase Ejecutivo se puede acceder al dato miembro sueldo, mientras que si se declara private no.

Up-casting y Down-casting

Siguiendo con el ejemplo de los apartados anteriores, dado que un Ejecutivo ES un Empleado se puede escribir la sentencia:

```
Empleado emp = new Ejecutivo("Máximo Dueño" , 2000);
```

Aquí se crea un objeto de la clase Ejecutivo que se asigna a una referencia de tipo Empleado. Esto es posible y no da error ni al compilar ni al ejecutar porque Ejecutivo es una clase derivada de Empleado. A esta operación en que un objeto de una clase derivada se asigna a una referencia cuyo tipo es alguna de las superclases se denomina 'upcasting'.

Cuando se realiza este tipo de operaciones, hay que tener cuidado porque para la referencia emp no existen los miembros de la clase Ejecutivo, aunque la referencia apunte a un objeto de este tipo. Así, las expresiones:

```
emp.aumentarSueldo(3);   // 1. ok.
aumentarSueldo es de Empleado
emp.asignarPresupuesto(1500);   // 2. error de
compilación
```

En la primera expresión no hay error porque el método aumentarSueldo está definido en la clase Empleado. En la segunda expresión se produce un error de compilación porque el método asignarPresupuesto no existe para la clase Empleado.

Por último, la situación para el método toString es algo más compleja. Si se invoca el método:

```
emp.toString();   // se invoca el metodo
toString de Ejecutivo
```

el método que resultará llamado es el de la clase Ejecutivo. `toString` existe tanto para Empleado como para Ejecutivo, por lo que el compilador Java no determina en el momento de la compilación que método va a usarse. Sintácticamente la expresión es correcta. El compilador retrasa la decisión de invocar a un método o a otro al momento de la ejecución. Esta técnica se conoce con el nombre de dinamic binding o late binding. En el momento de la ejecución la JVM comprueba el contenido de la referencia `emp`. Si apunta a un objeto de la clase Empleado invocará al método `toString` de esta clase. Si apunta a un objeto Ejecutivo invocará por el contrario al método toString de Ejecutivo.

Operador cast

Si se desea acceder a los métodos de la clase derivada teniendo una referencia de una clase base, como en el ejemplo del apartado anterior hay que convertir explicitamente la referencia de un tipo a otro. Esto se hace con el operador de cast de la siguiente forma:

```
Empleado emp = new Ejecutivo("Máximo Dueño" ,
2000);
Ejecutivo ej = (Ejecutivo)emp; // se convierte
la referencia de tipo
ej.asignarPresupuesto(1500);
```

La expresión de la segunda línea convierte la referencia de tipo Empleado asignándola a una referencia de tipo Ejecutivo. Para el compilador es correcto porque Ejecutivo es una clase derivada de Empleado. En tiempo de ejecución la JVM convertirá la referencia si efectivamente `emp` apunta a un objeto de la clase Ejecutivo. Si se intenta:

```
Empleado emp = new Empleado("Javier Todudas" ,
2000);
Ejecutivo ej = (Ejecutivo)emp;
```

no dará problemas al compilar, pero al ejecutar se producirá un error porque la referencia `emp` apunta a un objeto de clase Empleado y no a uno de clas Ejecutivo.

La clase `Object`

En Java existe una clase base que es la raíz de la jerarquía y de la cual heredan todas aunque no se diga explicitamente mediante la clausula `extends` . Esta clase base se llama `Object` y contiene algunos métodos básicos. La mayor parte de ellos no hacen nada pero pueden ser redefinidos por las clases derivadas para implementar comportamientos específicos. Los métodos declarados por la clase `Object` son los siguientes:

```
public class Object {
     public final Class getClass() { . . . }
     public String toString() { . . . }
     public boolean equals(Object obj)
{ . . . }
     public int hashCode() { . . . }
```

```
    protected Object clone() throws
CloneNotSupportedException { . . . }
    public final void wait() throws
IllegalMonitorStateException,

InterruptedException { . . . }
    public final void wait( long millis) throws
IllegalMonitorStateException,

    InterruptedException {. . .}
    public final void wait( long millis, int
nanos) throws

IllegalMonitorStateException,

InterruptedException { . . . }
    public final void notify() throws
IllegalMonitorStateException { . . . }
    public final void notifyAll() throws

IllegalMonitorStateException { . . . }
    protected void finalize() throws Throwable
{ . . . }
}
```

Las cláusulas `final` y `throws` se verán más adelante. Como puede verse `toString` es un método de Object, que puede ser redefinido en las clases derivadas. Los métodos `wait` , `notify` y `notifyAll` tienen que ver con la gestión de threads de la JVM. El método finalize ya se ha comentado al hablar del recolector de basura.

Para una descripción exahustiva de los métodos de Object se puede consultar la documentación de la API del JDK.

La cláusula `final`

En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida. Para esto está la cláusula `final` , que tiene significados levemente distintos según se aplique a un dato miembro, a un método o a una clase.

Para una clase, `final` significa que la clase no puede extenderse. Es, por tanto el punto final de la cadena de clases derivadas. Por ejemplo si se quisiera impedir la extensión de la clase Ejecutivo, se pondría:

```
final class Ejecutivo {
. . .
}
```

Para un método, `final` significa que no puede redefinirse en una clase derivada. Por ejemplo si declaramos:

```
class Empleado {
     . . .
     public final void aumentarSueldo( int
porcentaje) {
          . . .
     }
     . . .
}
```

entonces la clase Ejecutivo, clase derivada de Empleado no podría reescribir el método aumentarSueldo, y por tanto cambiar su comportamiento.

Para un dato miembro, `final` significa también que no puede ser redefinido en una clase derivada, como para los métodos, pero además significa que su valor no puede ser cambiado en ningún sitio; es decir el modificador `final` sirve también para definir valores constantes. Por ejemplo:

```
class Circulo {
     . . .
     public final static float PI = 3.141592;
     . . .
}
```

En el ejemplo se define el valor de PI como de tipo `float` , estático (es igual para todas las instancias), constante (modificador `final`) y de acceso público.

Herencia simple

Java incorpora un mecanismo de herencia simple. Es decir, una clase sólo puede tener una superclase directa de la cual hereda todos los datos y métodos. Puede existir una cadena de clases derivadas en que la clase A herede de B y B herede de C, pero no es posible escribir algo como:

```
class A extends B , C ....   // error
```

Este mecanismo de herencia múltiple no existe en Java.

Java implanta otro mecanismo que resulta parecido al de herencia múltiple que es el de las interfaces que se verá más adelante.