# Programación II
## Herencia en Java (Parte3)

**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 "***up-casting***".

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 clase Ejecutivo.

**Otro Ejemplo de Up-Casting**

El upcasting permite tomar decisiones en tiempo de ejecución.

```
public class Dibujo {
	private Figura figura;
	public Dibujo(){  }

	// Ejecturamos esIsosceles solo en Triangulo!
	public void setFigura(String figuraSTR){
		if (figuraSTR == "triangulo"){
			figura = new Triangulo(figuraSTR);
			System.out.println(((Triangulo)figura).esIsosceles());
		}

		if (figuraSTR == "cuadrado")  figura =
					new Cuadrado(figuraSTR);
	}

}

public class Figura {
	private String nombre;
	public Figura(){  }

	public Figura(String nombre){
		this.nombre = nombre;
	}
}

public class Cuadrado extends Figura{
	public Cuadrado(String nombre){
		super(nombre);
	}
}

public class Triangulo extends Figura{

	public Triangulo(String nombre){
		super(nombre);
	}
	boolean esIsosceles(){
		return true;
	}
}

public class Test {
	public static void main(String[] args) {
		Dibujo d = new Dibujo();
		d.setFigura("triangulo");

	}
}
```

**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 ...
    public final void wait(long millis) throws ...
    public final void wait(long millis, int nanos) throws ...
    public final void notify() throws ...
    public final void notifyAll() throws ...
    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.

**Referencias:**

http://www.arrakis.es/~abelp/ApuntesJava/Herencia2.htm