**HERENCIA**

Muchas veces distintos objetos comparten campos y métodos que hacen aproximadamente lo mismo (por ejemplo almacenar y devolver un nombre del ámbito humano con el que se designa al objeto, como el título de un álbum de música, el título de un libro, el título de una película, etc.).



Por ejemplo en un proyecto que utilice objetos Taxi y objetos Autobús podríamos encontrarnos algo así:



**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.

**SUPER**

//Código de la clase profesor, subclase de la clase Persona ejemplo aprenderaprogramar.com

public class Profesor extends Persona {

    //Campos específicos de la subclase.

    private String IdProfesor;

    //Constructor de la subclase: incluimos como parámetros al menos los del constructor de la superclase

    public Profesor (String nombre, String apellidos, int edad) {

        super(nombre, apellidos, edad);

        IdProfesor = "Unknown";   } //Cierre del constructor

    //Métodos específicos de la subclase

    public void setIdProfesor (String IdProfesor) { this.IdProfesor = IdProfesor;   }

    public String getIdProfesor () { return IdProfesor;   }

    public void mostrarNombreApellidosYCarnet() {

        // nombre = "Paco"; Si tratáramos de acceder directamente a un campo privado de la superclase, salta un error

        // Sí podemos acceder a variables de instancia a través de los métodos de acceso públicos de la superclase

        System.out.println ("Profesor de nombre: " + getNombre() + " " +  getApellidos() +

         " con Id de profesor: " + getIdProfesor() ); }

} //Cierre de la clase

Los aspectos a destacar del código son:

a) La clase persona es una clase “normal” definida tal y como lo venimos haciendo habitualmente mientras que la clase Profesor es una subclase de Persona con ciertas peculiaridades.

b) Los objetos de la subclase van a tener campos nombre, apellidos y edad (heredados de Persona) y un campo específico IdProfesor. El constructor de una subclase ha de llevar obligatoriamente como parámetros al menos los mismos parámetros que el constructor de la superclase.

c) **El constructor de la subclase invoca al constructor de la superclase.** Para ello se incluye, obligatoriamente, la palabra clave super como primera línea del constructor de la subclase. La palabra super irá seguida de paréntesis dentro de los cuales pondremos los parámetros que requiera el constructor de la superclase al que queramos invocar. En este caso solo teníamos un constructor de superclase que requería tres parámetros. Si p.ej. hubiéramos tenido otro constructor que no requiriera ningún parámetro podríamos haber usado uno u otro, es decir, super(nombre, apellidos, edad) ó super(), o bien ambos teniendo dos constructores para la superclase y dos constructores para la subclase. Ejemplo:

|  |
| --- |
| En la superclase:                                public Persona() {                                                       nombre = "";                                                       apellidos = "";                                                       edad = 0; }                                                        public Persona (String nombre, String apellidos, int edad) {                                                       this.nombre = nombre;                                                       this.apellidos = apellidos;                                                       this.edad = edad;                   } |

**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.

**ABSTRACT**

Declarar una clase abstracta es distinto a tener una clase de la que no se crean objetos. En una clase abstracta, no existe la posibilidad. En una clase normal, existe la posibilidad de crearlos aunque no lo hagamos. El hecho de que no creemos instancias de una clase no es suficiente para que Java considere que una clase es abstracta. Para lograr esto hemos de declarar explícitamente la clase como abstracta mediante la sintaxis que hemos indicado. Si una clase no se declara usando abstract se cataloga como “clase concreta”. En inglés abstract significa “resumen”, por eso en algunos textos en castellano a las clases abstractas se les llama resúmenes. Una clase abstracta para Java es una clase de la que nunca se van a crear instancias: simplemente va a servir como superclase a otras clases. No se puede usar la palabra clave new aplicada a clases abstractas. En el menú contextual de la clase en BlueJ simplemente no aparece, y si intentamos crear objetos en el código nos saltará un error.

A su vez, las clases abstractas suelen contener métodos abstractos: la situación es la misma. Para que un método se considere abstracto ha de incluir en su signatura la palabra clave abstract. Además un método abstracto tiene estas peculiaridades:

a) **No tiene cuerpo** (llaves): sólo consta de signatura con paréntesis.

b) Su signatura **termina con un punto y coma.**

c) **Sólo puede existir dentro de una clase abstracta.** De esta forma se evita que haya métodos que no se puedan ejecutar dentro de clases concretas. Visto de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.

d) Los métodos abstractos **forzosamente habrán de estar sobrescritos en las subclases.** Si una subclase no implementa un método abstracto de la superclase tiene un método no ejecutable, lo que la fuerza a ser una subclase abstracta. Para que la subclase sea concreta habrá de implementar métodos sobrescritos para todos los métodos abstractos de sus superclases.

**REFERENCIAS**

**HERENCIA****PROTECTED****SUPER****FINAL****ABSTRACT**