

## Tema 5 Las clases en java

- 5.1.- Definición.
- 5.2.- Declaración. Ejemplos.
- 5.3.- Atributos static.
- 5.4.-Tipos de métodos. Sobrecarga de métodos.
- 5.6.- El uso de this.
- 5.7.- Tipos de clases
- 5.8.- Métodos static.
- 5.8.- Diseño de clases
- 5.9.- Trabajando con objetos
- 5.10 .- El garbagge collector.

### 5.1.- Definición de clase

- v Una clase es una descripción de la estructura y el comportamiento de los objetos que pertenecen a ella.
- v Es el elemento básico de la P.O.O.

### 5.2.- Declaración de una clase

```
[ calificadores ] class nombreClase [ extend nombre_clase ] {
    [ calificadores ] tipo nomVar1;
    [ calificadores] tipo nomVar2;
    .....
    [ calificadores ] tipo nomMétodo1 ( [ lista_de_argumentos] ) {
        cuerpo
    }
    [ calificadores ] tipo nomMétodo1 ( [ lista_de_argumentos] ) {
        cuerpo
    }
    [ calificadores ] tipo nomMétodo1 ( [ lista_de_argumentos] ) {
        cuerpo
    }
}
```

- Una declaración de este tipo indica que la clase no descende de ninguna otra, aunque en realidad todas las clases declaradas en un programa Java, descienden directa o indirectamente de la clase *Object*, que es la raíz de toda la jerarquía de clases en Java.

### Ejemplo. Definición básica

```
class Punto {
    double x; // abscisa del punto
    double y; // ordenada del punto
}
```

- **x** e **y** son variables asociadas a cada objeto de la clase punto que se pueda crear.
- Se denominan atributos o variables de instancia.

```
class PruebaPunto {
    public static void main (String arg[ ] ) {
        Punto p1 = new Punto ( ) ;
        Punto p2 = new Punto ( ) ;
        Punto vp[ ] = new Punto [ 100 ];
        p1.x = -1.0;   p1.y = -1.0 ;
        p2.x = 1.0;    p2.y = 1.0 ;
        double distanOrigen = Math.sqrt(p2.x*p2.x+p2.y*p2.y) ;

        System.out.println("Distancia al origen "+distanOrigen) ;
    }
}
```

### Nueva definición de la clase punto

```
class Punto {
    private double x; // abscisa del punto
    private double y; // ordenada del punto

    public Punto ( ) { }

    public void asignar (double abs, double ord){
        x= abs; y= ord;
    }

    public double distanOrigen() {
        return (Math.sqrt (x*x + y*y) );
    }
}
```

#### Comentarios.

- Toda la información declarada **private** es exclusiva del objeto e inaccesible desde fuera de la clase.
- En cualquier clase existe por defecto un método sin tipo cuyo nombre es el de la propia clase. Es **el método constructor**, que se utiliza para crear un nuevo objeto con el operador **new**.

<pre> public double abscisa () {     return x; }  public double ordenada () {     return y; } } // fin de la clase </pre>	<p>➤ Toda la información declarada <b>public</b> es accesible desde fuera de la clase. Por defecto variables y métodos son <b>public</b>.</p>
---------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------

### 5.3.- Declaración de atributos . Atributos static.

Existen 2 tipos generales de atributos:

- **Atributos de objeto.-** Son variables u objetos que almacenan valores distintos para instancias distintas de la clase (para objetos distintos).
- **Atributos de clase.-** Son variables u objetos que almacenan el mismo valor para todos los objetos instanciados a partir de esta clase. Para declarar un atributo de clase se utiliza la palabra reservada **static**. Si no se especifica lo contrario los atributos son de objeto y no de clase .

**Atributos static** - Ejemplo -

```

class Persona {
    static int numPersonas = 0 ; // atributo de clase
    String nombre ;           // atributo de objeto

    public Persona (String n) { // constructor
        nombre =n;
        numPersonas++;
    }
    public void muestra () {
        System.out.print ("Soy "+nombre);
        System.out.print("pero hay "+(numPersonas-1) + "personas más");
    }
}

class Grupo {
    public static void main(String args[] ) {
        Persona p1, p2, p3;
        // creamos tres instancias del atributo nombre y sólo una del atributo numPersonas

```

```

p1 = new Persona ("Pedro");
p2 =new Persona ("Juan");
p3 =new Persona ("Susana");
p2.muestra() ;
p1.muestra(); }

```

## 5.4.- Tipos de métodos

Los métodos según la función que realizan respecto al objeto , se pueden clasificar en:

- ⇒ Constructores. Permiten construir el objeto. Punto()
- ⇒ Modificadores. Permiten alterar el estado del objeto. Asignar( double, double )
- ⇒ Consultores. Permiten conocer, sin alterar, el estado del objeto.

```

class Punto {
    private double x; // abscisa del punto
    private double y ; // ordenada del punto

```

```

    public punto () { }

```

M. Constructor, crea el objeto

```

    public punto (double abs, double ord) {
        x=abs;
        y= ord;
    }

```

Es posible definir un método constructor, que además de crear el objeto, altere su estado, por ejem. para inicializarlo

```

    public void asignar (double abs, double ord){
        x= abs; y= ord;
    }

```

M. Modificador cambia los valores de las variables de instancia.

```

    public double distanOrigen() {
        return (Math.sqrt (x*x + y*y ) ;
    }

```

M. Consultor, nos informa de los valores de las variables de instancia.

```

    public double abscisa () {
        return x; }

```

```

    public double ordenada (){
        return y; }

```

```

} // fin de la clase

```

```
class Prueba {
public static void main ( String arg [ ] ) {
    Punto p = new Punto;
    Punto p2 = new Punto;
    p.y=5 ; // sería un error, por ser información privada.
    p.asignar( 1, 1);
    double dist =p.distOrigen();
    System.out.println( " Distancia :"+dist ) ;
}
}
```

- Para poder utilizar un método correspondiente a un objeto, utilizaremos la notación de punto.

### Sobrecarga de métodos.

Se denomina sobrecarga, a la definición de un mismo elemento (símbolo, identificador..) con distintos significados, y en función de cómo se utilice puede interpretarse su significado.

*Ejem. El operador +*

```
a+=5;
System.out.print("vale :"+a);
```

Java permite explícitamente la sobrecarga de métodos. Tiene una gran importancia en la P.O.O. , especialmente debido a su uso cuando hay herencia.

### Ejem. Sobrecarga de métodos.

```
class Punto {
    private double x; // abscisa del punto
    private double y ; // ordenada del punto

    public Punto () { }

    public void asignar (double abs, double ord) {
        x= abs;
        y= ord;
    }

    public double distanOrigen() {
        return (Math.sqrt (x*x + y*y ) ;
    }
}
```

```
class Punto {
    private double x; // abscisa del punto
    private double y ; // ordenada del punto
    public Punto () { } // primer constructor

    public Punto ( double abs , double ord ) { //segundo c.
        x= abs;
        y= ord ; }

    public Punto ( double coord ) { // tercer constr.
        x= coord;
        y= coord ;
    }
}
```

<pre> public double abscisa () {     return x; }  public double ordenada () {     return y; } } // fin de la clase </pre>	<pre> <i>public void asignar (double abs, double ord){</i>     x= abs;     y= ord;  }  <i>public double distanOrigen() {</i>     return (Math.sqrt (x*x + y*y ) ; }  <i>public double abscisa () {</i>     return x; }  <i>public double ordenada () {</i>     return y; } } // fin de la clase </pre>
---------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- En este ejemplo tenemos tres métodos constructores que sólo se diferencian por su argumento. El lenguaje seleccionará uno u otro, dependiendo del argumento que utilicemos en la llamada al método.

**Ejem.**

```

class Prueba {
    Punto p= new Punto( );
    Punto p1 = new Punto (1.0, -1.0);
    Punto p2 = new Punto (1.0);
    p.asignar(1.0 , -1.0) ;
    .....
}

```

### 5.5.- El uso de this.

Java incluye una referencia especial denominada **this** que se utiliza dentro de cualquier método para hacer referencia al objeto actual. Ejem.

<pre> class Punto {     private double x ;     private double y ;     ..... } </pre>	<pre> // usamos el segundo constructor, pero a sus parámetros también les llamamos x e y public punto (double x, double y) {     <b>this.x = x;</b>     <b>this.y = y ;</b> } </pre>
--------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 5.6.- Modificadores de clase : tipos

Son palabras reservadas que se anteponen a la declaración de la clase. Los modificadores posibles son:

- public
- abstract
- final

### public

- Cuando se crean varias clases que se agrupan formando un paquete (package), sólo las declaradas **public** pueden ser accedidas desde otro paquete.
- Toda clase public, debe ser declarada en un fichero fuente con el nombre de esa clase pública: NombreClase.java
- *En un fichero fuente puede haber más de una clase, pero **sólo una** con el modificador **public**.*

### abstract

- Las clases abstract no pueden ser instanciadas.
- Sirven para declarar subclases que deben redefinir los métodos declarados abstract.
- Los métodos de una clase abstract pueden no ser abstract. En este último caso, tampoco se podrán declarar objetos de una clase declarada como abstract
- Cuando existe algún método abstract, la clase debe ser declarada abstract, en caso contrario el compilador dará un error.

#### ○ **Ejemplo: abstract**

```
abstract class Animal {
    String nombre;
    int patas;

    public Animal (String n, int p ) {
        nombre= n;
        patas = p;
    }


    abstract void habla ();
}
```

```
class Perro extends Animal {
    // La clase Perro es una subclase de Animal
    String raza ;

    public Perro (String n, int p, String r ) {
        Super (n, p );
        raza = r;
    }

    public void habla () {
```

Llama al constructor de la Superclase



```

System.out.println ("Me llamo " +nombre+ " :GUAU " );
System.out.println ("Mi raza es " +raza );
    }
    // cierra el método

```

```

} // cierra la clase

```

```

class Gallo extends Animal {    // La clase Gallo es una subclase de Animal
    public Gallo (String n, int p) {
        Super (n, p);
    }

    // Redefinimos habla para poder instanciar objetos de la clase Gallo.

    public void habla () {
        System.out.println("Soy gallo, me llamo "+nombre );
        System.out.println ("Kikirikiiiiiii " );
    }
}

```

```

class EjemAbstracta {
    public static void main ( String argm[ ] ) {
        Perro toby ;    // declaramos el objeto toby de la clase Perro
        toby = new Perro ( "Toby", 4 ,"San Bernardo");
        /* instanciamos el objeto toby, para que reciba las parámetros y métodos de la clase Perro. */
        Gallo kiko = new Gallo ("Kiko ", 2 ) ;
        kiko.habla();
        System.out.println() ;
        toby.habla();
    }
}

```



**final**

- Una clase declarada **final** impide que pueda ser superclase de otras clases. Ninguna clase puede heredar de una clase **final**.
- A diferencia del abstract, pueden existir en la clase métodos **final**, sin que la clase que los contiene sea final.
- Una clase puede ser a la vez:
  - public abstract
  - public final

**Resumen**

- **Declaración de una clase:** [ **Modificadores** ] class NombreClase { }
- **Modificadores de clase**
  - public** **Fichero fuente con el nombre de la clase.**
  - abstract** No se podrán crear objetos de esta clase.
  - Final** No puede ser superclase de otras. No se puede heredar.

**5.7.- Método static.**

Igual que las variables de clase, los métodos con el modificador static, son propios de la clase. Tienen las mismas reglas de visibilidad que las variables de clase, es decir, son públicos por defecto, y cuando deben utilizarse fuera de la clase donde se han definido, hay que utilizar el punto y el nombre de la clase donde están declarados.

## 5.8.- Diseño de clases.

Los pasos a seguir para definir una clase son:

1. Determinar exactamente el comportamiento que ofrecerá la clase. Dicho comportamiento vendrá determinado por los métodos que haga públicos la clase.
2. Enunciar cómo se utilizará la clase. Esto es, explicitar la cabecera de cada uno de los métodos. Al conjunto de dichas cabeceras, se le denomina *Interfaz* o *signatura* de la clase.
3. Escribir un programa sencillo que haga uso de la clase.
4. Escribir la implementación de la clase, explicitando las estructuras de datos que le darán soporte, los distintos atributos pertenecientes a la misma y sus tipos.

Hay que tener en cuenta, como en cualquier proceso de diseño, que es posible que alguna decisión tomada en una fase posterior, haga necesario alterar decisiones tomadas en fases preliminares.

## 5.9.- Trabajando con Objetos

El lenguaje Java es un lenguaje orientado a objetos, por lo que se puede decir que programar en Java consiste en “escribir las definiciones de las clases y utilizar esas clases para crear objetos” de forma que se represente adecuadamente el problema que se desea resolver.

El lenguaje Java posee un gran número de clases, objetos y métodos predefinidos, por lo que no es necesario reinventarlos, basta con utilizarlos cuando se necesiten.

Hasta el momento, se han utilizado los diferentes tipos primitivos del lenguaje Java (numéricos, carácter y lógico), junto con otros tipos que representaban información agregada, como las cadenas de caracteres, los arrays y los ficheros.

En Java, todos los tipos que no sean ninguno de los primitivos, esto es: int, long, double, short, char, boolean y byte, son tipos referencia.

Una variable referencia en Java es una variable que guarda, entre otras cosas, la dirección de memoria en la que se almacena un objeto. Cualquier objeto en Java se representa mediante una variable referencia.

*Una variable que represente a un objeto sólo almacena una referencia a la situación en memoria de dicho objeto.*

Las referencias, o direcciones, a los objetos son asignadas automáticamente por el compilador, y permanecen inaccesibles al usuario, lo que quiere decir que en un programa en Java no pueden existir operaciones explícitas de manejo de referencias.

Una referencia en Java es un mecanismo del compilador relativo al almacenamiento de los elementos que tienen estructura (los objetos). Debido a ello no existen en Java referencias a valores pertenecientes a tipos primitivos; por lo que, por ejemplo, la única forma de tener una referencia a un valor de tipo `int` o `double` consiste en *envolverlo en un objeto*. Esta última solución la adopta el propio lenguaje, predefiniendo para los tipos básicos las que se conocen como *clases envoltorio*, por ejemplo: las clases *Integer* o *Double*.

En particular, los únicos operadores permitidos para manipular los tipos referencia son las asignaciones vía el operador `=` y las comparaciones, mediante `==` y `!=`. Además, cuando una referencia no señala a ningún objeto entonces almacena la denominada referencia nula o `null`. La constante `null` es única y se utiliza para cualquier objeto (se dice que está sobrecargada).

**Objetos y referencias.** La diferencia principal entre valores primitivos y valores referencia consiste en que las variables que representan a los primeros mantienen el valor de los mismos mientras que las que representan a los segundos mantienen una referencia a su posición real en memoria. Todo ello implica un comportamiento distinto entre ambos tipos de variables que se refleja en su diferente manipulación.

**Declaración de variables.** En la forma de declarar los objetos existe una diferencia importante. La declaración del objeto no genera ese mismo objeto.

Por ejemplo, la siguiente secuencia en Java que declara y crea un objeto de tipo botón (`Button`):

```
Button boton;           //boton vale null
boton = new Button();    //boton referencia al objeto creado
```

Cuando se desea utilizar un nuevo objeto de cierto tipo, es necesario crearlo explícitamente. Hasta el momento de su creación el objeto no existe y cualquier intento de referenciarlo antes de dicho momento provocaría un error en tiempo de ejecución.

**El operador “.”** .Asociados a los objetos pueden existir operaciones o métodos que se aplicarán a los mismos.

El operador *punto* se emplea en dichos casos para seleccionar el método específico que se desee utilizar sobre el objeto en curso. En el ejemplo anterior, ahora ligeramente extendido, un Panel es un objeto gráfico representable en el que se sitúan elementos tales como botones, menús, etc.

```
Panel p = new Panel() ;      //Creación de un "Panel" p
Button boton ;              // boton vale null
boton = new Button() ;      // boton referencia al objeto creado
boton.setLabel("Ejemplo"); // Se pone etiqueta al botón
p.add(boton);               //Se añade el botón al Panel
```

Si, cuando se ejecuta un método asociado a un objeto, no existe este último (porque tiene el valor null), se produciría una excepción: **NullPointerException**.

Los métodos *setLabel()* y *add()*, se utilizan para escribir una etiqueta en un botón, y para añadir un elemento a un panel. Se encuentran definidos en las clases Button y Panel, respectivamente. Otro aspecto relevante es que muchos objetos pueden construirse dándoles un valor inicial, así:

```
Panel p = new Panel();      //Creación de un "Panel" p
Button boton = new Button("Ejemplo");
p.add(boton);               //Se añade el botón al Panel
```

**La asignación.** Dadas dos variables cualesquiera de tipos compatibles v1 y v2 la operación

`v1 = v2;` Asignación a v1 el valor de v2 reemplaza el contenido de v1 con el de v2, tanto si ambas pertenecen a uno de los tipos primitivos como si se trata de variables referencia. Pero, en el último caso, la asignación significa tan sólo un reemplazamiento de las referencias correspondientes, no del contenido referenciado por las mismas.

Ejemplo:

```
Objeto oB1 = new Objeto(); //oB1 referencia a un Objeto
Objeto oB2 = oB1;          //oB1 y oB2 referencian al Objeto primero
```

Tras la ejecución de las instrucciones, se tiene un único objeto referenciado por las dos variables oB1 y oB2 y, por lo tanto, se tiene un mismo objeto al que se puede nombrar de dos formas distintas: oB1 y oB2.

En el ejemplo siguiente, se pierde la referencia al segundo objeto:

```
Objeto oB1 = new Objeto(); //oB1 referencia a un Objeto
Objeto oB2 = new Objeto(); //oB2 referencia a otro Objeto
```

```
oB2 = oB1;                //oB1 y oB2 referencian al Objeto primero
```

Igual que antes, oB1 y oB2 nombran al mismo objeto. Pero ahora nada referencia al objeto creado en segundo lugar. En una situación así, cuando ninguna variable referencia a un objeto, se dice que dicho objeto está desreferenciado.

Ejemplo . Se trata de crear dos botones en cierto panel ya definido p:

```
// el Panel p ya ha sido creado
Button bot1 = new Button("BOT1");
Button bot2 = bot1;
bot2.setLabel("BOT2");
p.add(bot1);
p.add(bot2);
```

Observa que tan sólo se ha creado un botón, por lo tanto, tras la asignación efectuada en la segunda línea se tienen dos variables que referencian un mismo objeto, el creado en la línea primera. Tanto bot1 como bot2 representan un único objeto. La operación setLabel() se efectúa en cada caso sobre un único objeto, por lo que cuando ambos botones se añadan en la componente gráfica (el Panel) aparecerán con el mismo rótulo (BOT2).

La consecuencia que se puede extraer del ejemplo anterior es que es posible operar sobre un objeto utilizando cualquiera de los nombres de variables que lo representen. Y que, por supuesto, se tiene que ser cuidadoso cuando se dan situaciones de referenciación múltiple.

**Copia de objetos.** Puede parecer que, ya que la asignación entre objetos sólo supone una copia de las referencias, es imposible efectuar una copia de los objetos como tales. Sin embargo, esto no es cierto. En primer lugar, si la estructura del objeto es conocida y accesible, entonces es posible realizar dicha copia mediante una copia individual, elemento a elemento, de cada uno de los elementos del tipo primitivo correspondiente. Así, por ejemplo: Copia de un vector vec1 de N valores de tipo doublé. Para ello se debe crear un nuevo vector:

```
double vec2[] = new double[N];
for (int i=0; i<N ; i++)
    vec2[i] = vec1[i];
```

Y, en segundo lugar, existe un método, denominado **clone( )**, definido para cualquier objeto que permite efectuar dicha copia. Este método (clone) hace un uso explícito de new. Así, para el caso anterior: copia de un vector vec1 de N valores de tipo double

```
double vec2[] = (double []) vec1.clone();
```

El uso del método **clone** ha exigido una operación explícita de transformación de tipos (casting). Ello es así porque clone sólo devuelve una referencia a un elemento sin estructura o tipo conocido. El casting se la otorga.

**Paso de parámetros.** Como se sabe, en Java el paso de parámetros es siempre por valor. Ello quiere decir que cuando el parámetro de cierto método es un objeto, representado por tanto, mediante una referencia, entonces el valor que se transmite al método es la referencia a la zona de memoria donde se encuentra el objeto, no el del objeto en sí.

Naturalmente, cualquier modificación que se realice a través de la referencia al objeto original, implica una modificación del valor del objeto.

**El operador ==** . En los tipos primitivos el operador == es cierto o falso según sean o no iguales los valores de las variables que se comparan.

Cuando este operador se aplica a variables que referencian a objetos devolverá cierto o falso según sean o no iguales las referencias a dichos objetos. Por lo tanto, si se aplica el operador a dos objetos distintos, pero que contienen la misma información, el resultado que se obtendrá será false.

En Java, los contenidos de objetos distintos pueden compararse entre sí mediante el método equals para el que ya se ha visto algún uso con valores de tipo String. Sin embargo, equals( ) para algunos objetos es equivalente al test ==. Ocasionalmente, como en el caso de los Arrays pueden existir métodos específicos, como muestra el siguiente ejemplo:

```
double v1[] = new double[100];
for (int i=0; i<100 ; i++)
    v1[i]=i;
double v2[] = (double []) v1.clone();
double v3[] = v2;

if (Arrays.equals(v1,v1)) System.out.println("v1 igual v1");
if (Arrays.equals(v1,v2)) System.out.println("v1 igual v2");
if (Arrays.equals(v2,v3)) System.out.println("v1 igual v3");
// cada una de las comparaciones anteriores devuelve true.
```

Naturalmente, si se efectúa en las condiciones anteriores la comparación:

`if (v1 == v2) { ....` se evaluará a false, ya que v1 y v2 contienen referencias a distintas zonas de memoria.

### 5.10.- El garbage collector.

Cuando para un objeto dado, creado en algún momento de la ejecución de un programa no existe ninguna variable que lo referencie entonces decimos que dicho objeto está desreferenciado, lo que quiere decir que no es posible volver a operar con el.

En el siguiente ejemplo ya mostrado:

```
Objeto oB1 = new Objeto(); //oB1 referencia a un Objeto
Objeto oB2 = new Objeto(); //oB2 referencia a otro Objeto
oB2 = oB1;                //oB1 y oB2 referencian al Objeto primero, nada referencia al
                           objeto segundo
```

Tras la tercera instrucción toda referencia al objeto que se creó utilizando oB2 se ha perdido. Dicho objeto ya no será accesible.

Para evitar el uso innecesario de memoria, los lenguajes de programación introducen operaciones explícitas para informar al sistema de que una zona de memoria determinada (por ejemplo, la ocupada por un objeto) no está referenciada y, por lo tanto, es susceptible de ser reutilizada. En Java, cuando un objeto está desreferenciado, la memoria que consume se reclama automáticamente por un elemento denominado recogedor de basura ("garbage collector"). Su funcionamiento suele ser automático, aunque también puede ser ejecutado deliberadamente utilizando el método estático `System.gc()`.