PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA (I)

El lenguaje Java es totalmente orientado a objetos, esto significa que podemos aplicar en una aplicación Java todas las características y beneficios que proporciona este modelo de programación, entre ellas, la encapsulación, la herencia y el polimorfismo.

Vamos a estudiar todos estos conceptos clave en los que se basa la POO y su aplicación en Java. Se trata de conceptos cruciales, no sólo por su aplicación directa en el desarrollo de las aplicaciones, sino también porque están presentes en todo el conjunto de clases de Java.

Los distintos temas a tratar son:

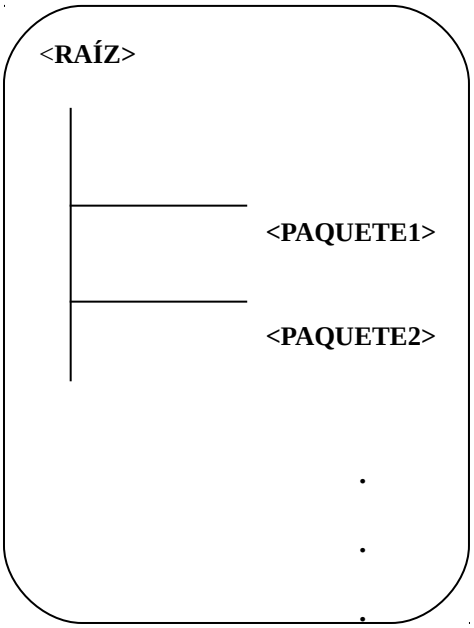
- Empaquetado de clases
- Modificadores de acceso
- Encapsulación
- Sobrecarga de métodos
- Constructores
- Herencia
- Sobrescritura de métodos
- Clases abstractas e interfaces
- Polimorfismo

1. EMPAQUETADO DE CLASES

La organización de las clases en paquetes facilita el uso de las mismas desde otras clases. Por ello, resulta también recomendable utilizar esta técnica en el desarrollo de nuestras propias clases.

Si queremos crear una estructura de paquetes en la que vamos a incluir las clases que vamos a definir, debemos proceder como sigue:

Creación de los directorios: Un paquete no es más que un directorio en el que estarán ubicados los archivos **.class** con las clases compiladas, así pues, lo primero que habrá que hacer es crear esos directorios dentro del directorio de trabajo. Todas las clases, organizadas en sus correspondientes paquetes, van a estar localizadas en este directorio de trabajo.



Empaquetado de las clases: Una vez creados los directorios, procederemos al empaquetado de las clases, para ello hemos de utilizar la sentencia *package* en el archivo de código fuente de la clase. La sintaxis de utilización de *package* es la siguiente:

package nombre_paquete;

Esta sentencia debe ser la primera instrucción del archivo .java, antes incluso que las sentencias import, y afectará a todas las clases existentes en el archivo.

Compilación: Finalmente, situaremos cada uno de los archivos de código fuente de las clases (java) en su subdirectorio correspondiente, según el paquete al que vayan a pertenecer sus clases. Al compilar, ya se crean los ficheros .class en el directorio correspondiente.

Ejemplo:

Supongamos que vamos a definir una clase para la obtención de mensajes de texto y queremos que esta clase pueda ser utilizada por otras clases que vayan a ser creadas en un futuro.

Hemos decidido que esta clase se va a ubicar en un paquete llamado ***pjava.***

En el directorio *pjava* situaremos el archivo de código fuente de la clase Ejemplo.java, cuyo listado se muestra a continuación:

```
package pjava;

public class Ejemplo {
	public String getMensaje() {
		return "hola";
	}
}
```

Una vez compilado, ya podemos hacer uso de *Ejemplo* desde cualquier otra clase que vayamos a crear, independientemente de donde esté situada. Eso sí, **para referirse a la clase** *Ejemplo* desde otro lugar habrá que utilizar el nombre cualificado de la misma (pjava.Ejemplo), o bien importarla **mediante la sentencia** *import*

```
:
import pjava.Ejemplo;

public class OtraClase {
	public void imprimirMensaje() {
		Ejemplo ej = new Ejemplo();
		System.out.println(ej.getMensaje());
	}
}
```

En Eclipse se haría así:

- Creamos un proyecto.
- Creamos un paquete dentro.
- Dentro del paquete creamos la clase.

```
package pjava;

public class Ejemplo {
	public String getMensaje() {
		return "Hola";
	}
}
```

En Eclipse, cuando se salvan los cambios, se compila el código, por lo que no hace falta realizar ese paso.

- En el paquete por defecto y dentro del mismo proyecto creamos dos clases:

```
import pjava.Ejemplo;

public class OtraClase {
	public void imprimirMensaje() {
		Ejemplo ej = new Ejemplo();
		System.out.println(ej.getMensaje());
	}
}
```

```
public class Probar {
	public static void main(String[] args) {
		OtraClase miMensaje = new OtraClase();
		miMensaje.imprimirMensaje();
	}
}
```

2. MODIFICADORES DE ACCESO

Los modificadores de acceso se utilizan para definir la visibilidad de los miembros de una clase (atributos y métodos) y de la propia clase.

En Java existen cuatro modificadores de acceso que, ordenados de menor a mayor visibilidad, son:

- **`private`** Cuando un atributo o método es definido como *private*, su uso está restringido a la propia clase, lo que significa que solamente puede ser utilizado **en el interior de su misma clase**. Este modificador puede ser aplicado a métodos y atributos, **pero no a una clase de primer nivel**, ya que sería una clase inútil pues ninguna otra podría acceder a ella. En cambio, sí es posible utilizar el modificador de acceso private en una clase interna o anidada, esto es , una clase que se declara dentro de otra.

- **(ninguno)** La no utilización de modificador de acceso, proporciona al elemento lo que se conoce como el *acceso por defecto.* Si un elemento (clase, método o atributo) tiene acceso por defecto, únicamente **las clases de su mismo paquete** tendrán acceso al mismo. El siguiente código aclara el funcionamiento de este modificador:

```
package paquete1;
class ClaseA {
        void metodo1() {
        .
                System.out.println("En metodo1 claseA");

        .
        .
        }
}

public class ClaseB {
        void metodo2() {
                /*correcto, ambas clases están en
                el mismo paquete*/
                ClaseA ca = new ClaseA();
                ca.metodo1();
        }
}
```

```
package paquete2;
public class ClaseC {
        void metodo3() {
                /*error, las clases están en paquetes
                diferentes*/
                ClaseA ca = new ClaseA();
        }

        void metodo4() {
          paquete1.ClaseB cb = new paquete1.ClaseB();
          /*error, metodo2() no es visible
          desde ClaseC */
          cb.metodo2();
        }
}
```

- **`protected`** Se trata de un modificador de acceso empleado en la herencia, De momento, basta decir que un método o atributo definido como *protected* en una clase **puede ser utilizado por cualquier otra clase de su mismo paquete y además, por cualquier subclase de ella, independientemente del paquete en que ésta se encuentre.** Una clase no puede *ser protected*, sólo sus miembros; la excepción son las clases anidadas, que sí se pueden declarar como protected.

- **`public`** El modificador *public* ofrece el máximo nivel de visibilidad. Un elemento (clase, método o atributo) *public* será visible desde cualquier clase, independientemente del paquete en que se encuentre.

El cuadro resume la aplicabilidad de los modificadores de acceso sobre los distintos componentes de una clase.

	private	(default)	protected	public
clase	NO	SI	NO	SI
método	SI	SI	SI	SI
atributo	SI	SI	SI	SI
variable local	NO	NO	NO	NO

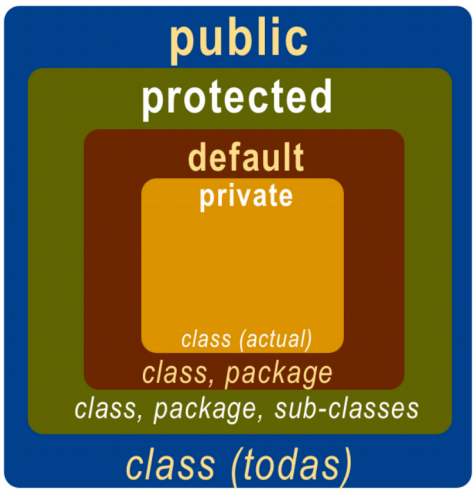
Aplicación de modificadores de acceso a los componentes de una clase

Los modificadores de acceso se utilizan en las definiciones de:

- Clases: sólo se puede utilizar public o default (sin modificador).
- Atributos: se permite cualquiera de los cuatro.
- Métodos: se permite cualquiera de los cuatro.

Resumen del funcionamiento de los modificadores de acceso en Java:

Modificador	La misma clase	Mismo paquete	Subclase	Otro paquete
`private`	Sí	No	No	No
`default`	Sí	Sí	No	No
`protected`	Sí	Sí	Sí	No
`public`	Sí	Sí	Sí	Sí



Además, y en combinación con los anteriores, pueden utilizarse los modificadores: **`static`** y **`final`**, que veremos a continuación:

Static

A. Métodos estáticos

Existen casos en los que nos encontramos con clases cuyos métodos **no dependen en absoluto de los atributo de la clase**, y en todo caso de los parámetros de los métodos. Por ejemplo, la clase java.lang.Math:

- Su método `round()` recibe un número decimal y lo devuelve redondeado.
- Su método `sqrt()` recibe un número y devuelve su raíz cuadrada.
- Su método `min()` recibe dos números y devuelve el menor.

Son métodos que parecen no pertenecer a una entidad concreta. Son genéricos, globales, independientes de cualquier estado del objeto.

¿Tiene sentido instanciar un objeto para ejecutar algo que no depende de nada de dicho objeto?

La respuesta es no. Y para ello contamos en Java con los métodos estáticos. Están asociados a una clase solamente desde un punto de vista organizativo.

Para definir un método estático utilizamos la palabra clave: `static`. La sintaxis de la declaración es la siguiente:

modificador_acceso static tipo_retorno nombre([tipo parametro,..]) {
}

Ejemplo:

```
public static void miMetodo() {
}
```

Para ejecutar por tanto un método estático no hace falta instanciar un objeto de la clase. Se puede ejecutar el método directamente sobre la clase.

Ejemplo:

```
int a = Math.min(10,17);
```

Una clase puede perfectamente mezclar métodos estáticos con métodos convencionales. Un ejemplo clásico es el método main:

```
public static void main(String[] args) {
      …
}
```

Hay ciertas reglas que hay que tener en cuenta en el uso de métodos estáticos:

- Un método estático no puede acceder a un atributo de instancia (no estático).
- Un método estático no puede acceder a un método de instancia (no estático).
- Pero desde un método convencional si que se puede acceder a atributos y métodos estáticos.

B. Atributos estáticos

Los atributos estáticos (o variables estáticas) son atributos cuyo valor es compartido por todos los objetos de una clase. Algunos autores también los definen como campos de clase.

Para definir un atributo estático utilizamos la palabra clave: `static`. La sintaxis de la declaración es la siguiente:

modificador_acceso static tipo nombre [= valor_inicial];

Ejemplo:

```
public static int contador = 0;
```

Hay que tratarlos con cuidado puesto que son fuente de problemas difíciles de detectar. Como todos los objetos de una misma clase comparten el mismo atributo estático, si un objeto 'a' modifica el valor del atributo estático, cuando el objeto 'b' vaya a usar dicho atributo, éste va a tener el valor modificado.

Recordemos que sin embargo los atributos convencionales (de instancia) son propios de cada objeto.

Los atributos estáticos son cargados en memoria cuando se carga la clase, siempre antes de que:

- se pueda instanciar un objeto de dicha clase.
- se pueda ejecutar un método estático de dicha clase.

Para usar un atributo estático no hace falta instanciar un objeto de la clase.

Ejemplo:

```
System.out.println("Hola");
// out es un atributo estático de la clase java.lang.System
```

Ejemplo:

```
public class Jugador {
	public static int contador = 0;
	private String nombre = null;

	public Jugador(String n){
		nombre = n;
		contador++;
	}
}

public class Test {
	public static void main(String[] args) {
		System.out.println(Jugador.contador);
		Jugador uno = new Jugador("Pedro");
		System.out.prinln(Jugador.contador);
	}
}
```

La ejecución del programa anterior muestra la siguiente salida:

```
0
1
```

C. Bloques de código estáticos

Los bloques de inicialización estáticos son trozos de código que se ejecutan una sola vez, al cargar una clase en memoria (no al instanciar objetos de esa clase), y siempre se ejecutará antes de cualquier constructor de instancia.

Permite inicializar los miembros estáticos finales de la clase (constantes de clase), ya que esto no se puede hacer desde los contructores de instancia. También se usa para inicializar miembros con valores calculados que requieren varias líneas.

Para definir un bloque de código estático utilizamos la palabra clave: `static`. La sintaxis de la declaración es la siguiente:

static { …. }

Ejemplo:

```
static {
      System.out.println("Hola");
}
```

Ejemplo:

```
public class BloqueEstatico {
      static {
            System.out.println("Bloque estático");
      }

      public BloqueEstatico() {
            System.out.println("Constructor");
      }
}
public class TestEstatico {
     public static void main(String[] args) {
            BloqueEstatico bs = new BloqueEstatico();
      }
}
```

Al ejecutar el programa anterior, se produce la siguiente salida:

```
Bloque estático
Contructor
```

Final

final es una palabra clave que modifica el funcionamiento de:

- Clases
- Atributos
- Métodos

A. Clases finales

Definiendo una clase como final conseguimos que ninguna otra clase pueda heredar de ella. Esto es particularmente útil a la hora de crear clases inmutables, como la clase String. Para definir una clase final utilizamos la palabra clave: `final`.

La sintaxis de la declaración es la siguiente:

modificador_acceso final class nombre_clase {
...
}

Ejemplo:

```
public final class MiClase {
}
```

Ejemplo: *intento de herencia de una clase final*

```
public final class ClasePadre {
}

public class ClaseHija extends ClasePadre {
}
```

Se produce un error porque `ClaseHija` no puede heredar de `ClasePadre` ya que esta es `final`.

B. Métodos finales

Definiendo un método como final conseguimos que dicho método no pueda ser sobreescrito por sus subclases.

Para definir un método como final utilizamos la palabra clave: `final`. La sintaxis de la declaración es la siguiente:

modificador_acceso final tipo_retorno nombre([tipo param,...]) {
...
}

Ejemplo:

```
public final int sumar(int param1, int param2) {
	return param1 + param2;
}
```

Ejemplo:

```
public class ClasePadre {
	public final int sumar(int a, int b)  {
		return a + b;
	}
}
public class ClaseHija extends ClasePadre {
	public int sumar(int a, intb) {
		return b + a;
	}
}
```

Se produce un error porque `ClaseHija` no puede sobreescribir el método `sumar()` de `ClasePadre` ya que está declarado como `final`.

C. Atributos finales

Definiendo un atributo como `final` conseguimos constantes. Es decir, una vez inicializados no se puede cambiar su valor.

Para definir un atributo como final utilizamos la palabra clave: `final`. La sintaxis de la declaración es la siguiente:

modificador_acceso final tipo nombre [= valor_inicial];

Ejemplo:

```
protected final boolean sw = true;
public final int i;
```

Ejemplo:

```
public class FinalTest {
    final int tmp1 = 3; // ya no se va a poder modificar tmp1
    final int tmp2;

    public FinalTest() {
          tmp2 = 42; // ya no se va a poder cambiar el valor de tmp2
    }

    public void realizarTarea(final int tmp3) {
          // ya no se va apoder cambiar tmp3 aquí dentro
    }

    public void realizarOtraTarea() {
          final int tmp4 = 7; // ya no se va a poder cambiar tmp4
          tmp4 = 5;
    }
}
```

D. Definición de constantes

Las constantes en Java se suelen definir mediante la combinación de las palabras clave: **`static`** y **`final`**. La sintaxis de la declaración es la siguiente:

modificador_acceso static final tipo nombre = valor;

Ejemplo:

```
public static final double PI = 3.141592653589;
```

Por convención, a la hora de programar, las constantes se suelen nombrar con todas las letras en mayúsculas.

Ejemplo: *Constantes ya existentes en las clases básicas:*

`java.lang.Math.PI` --> el número PI.
`java.lang.Math.E` --> el número E.

3. ENCAPSULACIÓN

Una clase está compuesta, por un lado, de métodos que determinan el comportamiento de los objetos de la clase y, por otro, de atributos que representan las características de los objetos de la clase.

Los **métodos** que se quieren exponer al exterior llevan el modificador de acceso **`public`**, mientras que los **atributos** suelen tener **acceso privado**, de modo que solamente puedan ser accesibles desde el interior de la clase.

Ésa es precisamente la idea de la encapsulación: **mantener los atributos de los objetos como privados y proporcionar acceso a los mismos a través de métodos públicos (métodos de acceso).** Esta filosofía de programación proporciona grandes beneficios, entre los que cabría destacar:

- Protección de datos "sensibles".
- Facilidad y flexibilidad en el mantenimiento de las aplicaciones.

3.1. Protección de datos

Imaginemos que tenemos que crear una clase que representa una figura geométrica, por ejemplo, un rectángulo. Dicha clase podría proporcionar diversos métodos para realizar cálculos sobre la figura, además de disponer de los atributos que la caracterizarían, como pueden ser alto y ancho.

Supongamos que desarrollamos la clase sin aplicar el concepto de encapsulación, proporcionando acceso público a los atributos:

```
public class Rectangulo {
        public int alto, ancho;
        //Métodos de la clase
        ....
}
```

Al utilizar esta clase desde cualquier otro programa e intentar asignar valores a los atributos, nada impediría al programador que va a realizar esa tarea hacer algo como esto:

```
Rectangulo r = new Rectangulo();
r.alto = -5;
```

Lógicamente, dimensiones con valores negativos no tienen sentido en una figura geométrica, además de provocar resultados incoherentes en la ejecución de los métodos de la clase.

A este hecho se le conoce también como corrupción de los datos, y una forma de evitarlo sería proteger los atributos del acceso directo desde el exterior mediante la encapsulación, forzando a que el acceso a dichos atributos se realice siempre de forma "controlada" a través de métodos de acceso, generalmente getters y setters.

En el ejemplo de la clase `Rectangulo`, la encapsulación de los atributos podría realizarse de la siguiente forma:

```
public class Rectangulo {
	private int alto, ancho;
	public void setAlto(int alto) {
		if (alto > 0)
		this.alto = alto;
	}
	public int getAlto() {
		return this.alto;
	}
	public void setAncho(int ancho) {
		if (ancho > 0)
		this.ancho = ancho;
	}
	public int getAncho() {
		return this.ancho ;
	}
	//Otros métodos de la clase
}
```

Se sigue el convenio *setNombre_atributo* para nombrar al método de acceso que permite la escritura del atributo y *getNombre_atributo* para el método de lectura.

La creación de objetos *Rectangulo* y asignación de valores a los atributos utilizando estos métodos, sería de la siguiente forma:

```
Rectangulo r = new Rectangulo();
r.setAlto(3);
r.setAncho(6);
```

Así, una instrucción como la siguiente:

```
r.setAlto(-4);
```

Provocaría que la variable `alto` permaneciese invariable, impidiendo que pueda tomar un valor negativo.

Gracias al control que se hace en el método de acceso antes de almacenar un valor en el atributo, se evita que dicho atributo se corrompa.

Obsérvese el uso de la palabra `this` para acceder a los atributos. **La palabra reservada `this` se utiliza en el interior de una clase para invocar a los métodos y atributos del propio objeto:**

```
this.metodo();
```

Sin embargo, su uso es redundante ya que los métodos y atributos propios pueden ser llamados directamente por su nombre sin necesidad de utilizar `this`. Tan sólo será necesario utilizar esta palabra reservada para invocar a un miembro del propio objeto cuando nos encontremos una variable local y un atributo con el mismo nombre, en cuyo caso, la manera de referirse al atributo será:

this.variable_atributo

3.2. Facilidad en el mantenimiento de la clase

Si una vez creada la clase queremos cambiar el criterio sobre los posibles valores que pueden tomar los atributos, podríamos modificar el código de los métodos de acceso y distribuir la nueva versión de la clase, sin que los programadores que la utilizan tengan que cambiar una sola línea de código de sus programas.

Por ejemplo, si se decide que el valor del atributo alto no puede ser inferior a 2, el método `setAlto()` quedaría:

```
public void setAlto(int alto) {
      if (alto > 1) {
            this.alto=alto;
      }
}
```

Los detalles de implementación quedan ocultos, manteniendo la interfaz (el formato y utilización del método no cambian) por lo que el código que hace uso de esta clase no tendrá que modificarse.

3.3. Clases de encapsulación (JavaBeans)

En muchos tipos de aplicaciones, puede resultar útil la creación de clases cuya única finalidad sea la encapsulación de una serie de datos dentro de la misma, datos que están asociados a una entidad (información de un empleado, de un libro, de un producto, etc.) y que conviene tenerlos agrupados en un mismo objeto de cara a facilitar su tratamiento.

A estas clases se las conoce como clases de encapsulación o **JavaBeans** (aunque este término puede utilizarse también en otros contextos) y, aparte de los campos y del constructor público por defecto, solamente dispone de métodos *set/get*.

El siguiente código corresponde a una clase de estas características:

```
public class Empleado {
      private String nombre;
      private String dni;

      public Empleado() {

      }

      public void setNombre(String nombre) {
            this.nombre = nombre;
      }

      public String getNombre() {
            return nombre;
      }

      public void setDni(String dni) {
            this.dni = dni;
      }

      public String getDni() {
            return dni;
      }
}
```

4. SOBRECARGA DE MÉTODOS

Otra de las ventajas que nos ofrece la POO orientada a objetos es poder tener en una **misma clase varios métodos con el mismo nombre,** a esto se le llama **sobrecarga** de métodos.

Ejemplos de sobrecarga nos hemos encontrado ya en las clases de uso general que se han visto hasta el momento. Un caso concreto es el los métodos `valueOf()` de la clase `String` para conversión de tipos básicos en cadenas de caracteres, donde tenemos una versión de este método para cada uno de los tipos básicos Java.

La gran ventaja de la sobrecarga de métodos es que, si tenemos varios métodos que van a realizar la misma operación (por ejemplo, convertir un tipo básico en una cadena), no necesitamos asignarle un nombre diferente a cada uno (con la consiguiente dificultad a la hora de aprenderlos y posible confusión), sino que podemos llamarlos igual a todos ellos.

Para que un método pueda sobrecargarse tiene que darse la siguiente condición:

Cada versión del método debe distinguirse de las otras en el número o tipo de parámetros.

El tipo de devolución puede ser o no el mismo.

Los siguientes son ejemplos válidos de sobrecarga:

```
public void calcular(int k) { ... }
public void calcular(String s) { ... }
public long calcular(int k, Boolean b) { ... }
```

En cambio, los siguientes métodos incumplen alguna de las condiciones de la sobrecarga y, por tanto, no podrán estar los tres en la misma clase:

```
public void calcular(int k) { ... }
public int calcular(int k) { ... }
// aunque cambia el tipo devuelto, la lista de
// parámetros es igual a la del anterior
public void calcular(int n) { ... }
// es idéntico al primero
```

La sobrecarga de métodos permite, por tanto, disponer de diferentes versiones de un método para llevar a cabo una determinada operación. A la hora de invocar a un método sobrecargado en un objeto, el compilador identificará la versión del método que se quiere invocar por los argumentos utilizados en la llamada. Dada la siguiente clase:

```
class Ejemplo {
        public void saludar() {
                System.out.println("Hola");
        }
        public void saludar(String nombre) {
                System.out.println("Hola " + nombre);
        }
}
```

Al ejecutarse la siguiente clase:

```
class Uso {
        public static void main(String[] args){
                Ejemplo ej = new Ejemplo();
                ej.saludar();
        }
}
```

Aparecerá en pantalla la frase: `Hola` porque se ha ejecutado la versión sin parámetros del método `saludar()`.

5. CONSTRUCTORES

5.1. Definición y utilidad

Para entender la utilidad de los constructores vamos a utilizar como ejemplo la clase `Punto`. Esta clase representa puntos geométricos caracterizados por dos coordenadas, `x` e `y`, donde a modo de ejemplo se ha añadido un método llamado `dibujar()` y cuya única función es mostrar en pantalla los valores de las coordenadas del punto. La implementación de esta clase sería la siguiente:

```
public class Punto {
      private int x,y;

      public int getX(){
         return x;
      }
      public int getY() {
         return y;
      }
      public void setX(int x) {
         this.x = x;
      }
      public void setY(int y) {
         this.y = y;
      }
      public void dibujar() {
          System.out.println("Las coordenadas son "+x+","+y);
      }
}
```

Si quisiéramos crear un punto a partir de esta clase y posteriormente dibujarlo (llamada al método `dibujar()`, deberíamos hacer algo como esto:

```
Punto pt = new Punto();
pt.setX(6);
pt.setY(lO);
pt.dibujar();
```

Como se ve, cada vez que se quiere crear un punto para hacer posteriormente alguna operación con el mismo, es necesario primero llamar explícitamente a los métodos `setX()` y `setY()` para asignar valores a las coordenadas del punto. Esto, además de resultar pesado en caso de tener muchos atributos, puede dar lugar a olvidos y, por tanto, a que ciertos atributos queden sin ser inicializados de manera explícita (tomarían un valor por defecto). Para evitar estos problemas, tenemos los **constructores.**

Un constructor es un método especial que es ejecutado en el momento en que se crea un objeto de la clase (cuando se llama al operador `new`). Podemos utilizar los constructores para añadir aquellas tareas que deban realizarse en el momento en que se crea un objeto de la clase, como por ejemplo, la inicialización de los atributos.

A la hora de crear un constructor, hay que tener en cuenta las siguientes reglas:

- El nombre del constructor debe ser el mismo que el de la clase.
- El constructor no puede tener tipo de devolución, **ni siquiera** `void`**.**
- Los constructores, como métodos que son, se pueden **sobrecargar,** lo que significa que una clase puede tener más de un constructor y por tanto distintas formas de inicializar sus atributos. En este sentido, se deben seguir las mismas reglas que se definieron para la sobrecarga de métodos.
- Toda clase debe tener, al menos, un constructor. En el caso de que no se declare un constructor explícitamente, el compilador le añadirá uno por defecto.

En el siguiente listado se muestra la clase **Punto** con dos constructores para la inicialización de las coordenadas.

```
public class Punto {
	private int x,y;

	public Punto(int x, int y) {
		this.x = x;
		this.y = y;
	}
	public Punto(int v) {
		this.x = v;
		this.y = v;
	}
	//resto del código de la clase
}
```

Ahora podemos hacer uso de estos constructores para hacer más cómoda la creación de un objeto **Punto**, pasándole las coordenadas del punto al constructor en la instrucción de creación del objeto:

```
Punto pt = new Punto(4, 6); // Llama al primer constructor
pt.dibujar();
```

Dado que el constructor almacena los números en los datos miembro **x** e **y**, la llamada a **dibujar()** en la instrucción anterior provocará que se muestre en pantalla el texto:

```
Las coordenadas son 4 y 6
```

Si las dos coordenadas fuesen a tener el mismo valor, podríamos haber optado por el segundo constructor:

```
Punto pt = new Punto(5); // x e y tomarán el valor 5
pt.dibujar();
```

5.2 Constructores por defecto

Según la cuarta de las reglas que hemos dado para la creación de constructores, toda clase debe tener al menos un constructor, pero, ¿qué sucede si creamos una clase sin constructores?

En este caso **el compilador de Java añadirá un constructor** a nuestra clase, denominado constructor por defecto, cuyo aspecto será:

```
public Nombre_Clase() {

}
```

Es decir, será un constructor sin parámetros y sin código, pero necesario para que la clase pueda compilar.

En el caso de la primera versión de la clase **Punto**, en la que no habíamos creado ningún constructor de manera explícita, el compilador Java añadirá implícitamente el siguiente constructor:

```
public Punto(){
}
```

Sin él, no sería posible crear objetos punto de la forma:

```
Punto pt = new Punto();
```

Así pues, siempre que se defina una clase sin constructores, el compilador añadirá uno por defecto sin parámetros y sin código. Hay que tener en cuenta que **este constructor por defecto será añadido por el compilador Java solamente si la clase carece de constructores.**

Cuando una clase cuenta con constructores, no se añadirá ningún constructor de forma implícita. En este caso, si la clase no dispone de un constructor sin parámetros, como sucede con la última versión de la clase Punto, una instrucción como ésta:

```
Punto pt = new Punto();
```

provocará un **error de compilación.** Este hecho es importante tenerlo en cuenta cuando se está desarrollando una clase, ya que si además de poder inicializar atributos en la creación de los objetos se desea ofrecer la posibilidad de crear objetos de la clase sin realizar ningún tipo de inicialización, será necesario codificar explícitamente un constructor por defecto en la clase. Así pues, para que la operación anterior pudiera realizarse, la clase Punto debería ser:

```
public class Punto {
	private int x,y;

	//constructor por defecto
	public Punto() {
	}

	public Punto(int x, int y) {
		this.x = x;
		this.y = y;
	}

	public Punto(int v) {
		this.x = v;
		this.y = v;
	}

	//resto del código de la clase
}
```