

Fundamentos de Programación

Tema 2: La programación orientada a objetos

Contenidos

1.- Introducción	2
2.- La programación orientada a objetos	3
3.- Clases y objetos	6
4.- Métodos constructores	11
5.- Métodos de instancia	13
6.- Documentación de las clases	17
7.- Excepciones	22
8.- Herencia	29
9.- Interfaces	33
10.- Métodos estáticos	38
11.- Generics	40
12.- Constantes	44
13.- Tipos de dato referencia	46

2.- La programación orientada a objetos

En el mundo real, cuando una persona debe resolver un problema o realizar una tarea, utiliza los objetos de su alrededor para llevarla a cabo.

Por ejemplo, si una persona tiene que construir una mesa, necesitará trozos de madera, una sierra, martillo, cola... además, deberá tener una hoja de instrucciones que le indique cómo se construye la mesa usando esas herramientas.



La programación orientada a objetos pretende trasladar esta idea a la programación de ordenadores: si las personas utilizan objetos en el mundo real, ¿por qué no utilizar **representaciones informáticas de los objetos** para realizar los programas de ordenador?

De esta forma, un programador podrá coger todos los objetos informáticos que necesite y utilizarlos para hacer su programa. Cada objeto tendrá sus propias características.

Los objetos informáticos son representaciones informáticas de objetos reales (una persona, un reproductor de música, un libro, una piedra, una bolsa que contiene otros objetos, un sprite de un videojuego...) o de cosas abstractas (una conexión con una base de datos, un polinomio, un programa de ordenador, una ecuación de 2º grado...)

Como veremos más adelante, cualquier cosa real o imaginaria, podrá informatizarse y podremos usarla para realizar programas de ordenador.

Por tanto, el programador simplemente tendrá que coger los objetos que necesite y utilizarlos en su programa. Los objetos obedecerán al programador, y de esta forma, el programa realizará su misión.

Los objetos van a constituir nuevos tipos de datos, que denominaremos **tipos de dato referencia**, y que tendrán propiedades diferentes a los tipos básicos que estudiamos en el tema 2. De esta forma, podremos tener tipos de dato referencia llamados "Radio", "Coche", "Ecuación", etc.

Con los objetos, un programador puede realizar básicamente dos cosas:

- **Darles órdenes:** Cada objeto va a tener una serie de órdenes que podremos darle. Por ejemplo, si tenemos una radio, algunas órdenes que podríamos darle sería *“sintoniza la emisora 103.90 MHz”*, *“sube el volumen al 80%”*, o *“apágate”*. Cada vez que damos una orden al objeto, éste nos obedecerá, realizando la acción que le ha sido ordenada.



- **Hacerles preguntas:** Podremos pedir al objeto información, y éste nos dará una respuesta, que **deberemos guardar en una variable**¹. Por ejemplo, si tenemos una cuenta corriente de un banco, podríamos preguntarle: *“¿Cuánto dinero hay en la cuenta?”*. La respuesta a esta pregunta es un número que puede tener decimales, y por tanto, dicha respuesta debería ser guardada en una nueva variable de tipo double. Otra pregunta podría ser: *“¿Cómo se llama el titular de la cuenta?”*. En este caso, la respuesta es un texto, por lo que se guardaría en una variable de tipo String.

En muchos casos, la respuesta a una petición puede ser otro objeto. Por ejemplo, si tenemos un objeto “Persona” podemos decirle: *“Dame tu DNI”*. La respuesta es un objeto con su carnet de identidad.



Como podemos ver, cada objeto va a tener su propia forma de uso. Por este motivo, los objetos vienen acompañados de unas **hojas de instrucciones** que indican cómo deben utilizarse. En ella encontraremos las órdenes que podemos darles y las preguntas que podemos hacerles. Las instrucciones para manejar cada objeto las podremos encontrar en formato papel o en formato de página web, según como las distribuya su fabricante.

Por último, indicamos que los objetos, una vez que han sido programados, pueden reutilizarse en cualquier programa. Un programador puede construir sus propios objetos o bien puede acudir a una **librería** con objetos realizados por otros programadores.

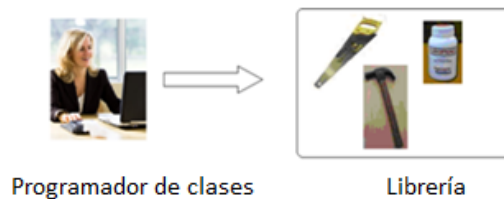
¹ La **principal causa de suspensos** es hacer una pregunta a un objeto y no guardar su respuesta en una variable. De esta forma, la respuesta se pierde y es imposible continuar haciendo el programa.

Ejercicio 1 : Elige el objeto que tú quieras del mundo real e inventa tres órdenes que podrías darle y tres preguntas que podrías hacerle. Por cada pregunta, deberás indicar también cuál es el tipo de dato de la respuesta que proporciona.

2.1.- Perfiles del programador

En programación orientada a objetos existen dos perfiles de programador, según la función que un programador esté realizando en un momento dado:

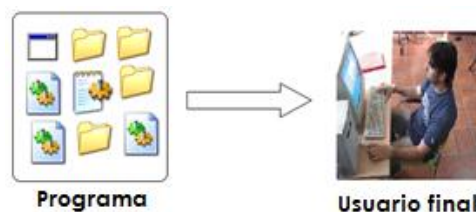
- **Programador de clases:** Es un programador que crea tipos de dato referencia y los guarda en **librerías** para que otros programadores puedan usarlos en sus programas. En Java, las librerías están divididas en carpetas llamadas **paquetes**, que sirven para poner un poco de orden y agrupar su contenido, de la misma forma que organizamos los archivos en carpetas en un disco duro.



- **Programador de aplicaciones:** Es un programador que realiza programas utilizando tipos de dato referencia que ya están hechos.



Finalmente todavía habría un último paso que sería el usuario final de la aplicación:



Como podemos imaginar, la tarea más complicada de todo el proceso suele ser la de diseñar y construir los objetos necesarios para la aplicación. Una vez que los objetos ya están creados, realizar el programa suele ser una tarea bastante más sencilla.

En este tema adoptaremos el perfil de programador de aplicaciones, y por tanto nos centraremos en realizar programas usando objetos que ya han hecho otras personas. Para ello primero aprenderemos a usar objetos de forma general, interpretando las instrucciones de uso que publican sus programadores.

3.- Clases y objetos

Comprender los conceptos de **clase** y **objeto** es el punto de partida para aprender programación orientada a objetos, ya que todo gira en torno a ellos. Es muy importante comprenderlos muy bien y dominar toda la terminología con mucha soltura, puesto que es el “abc” de la programación actual.

3.1.- Las clases

Comenzamos definiendo el concepto más importante de la programación orientada a objetos:

Una **clase** es un **tipo de dato** que informatiza una colección de objetos similares

Por tanto, una **clase** es el concepto abstracto, que agrupa un conjunto de objetos de iguales características. Por ejemplo, podemos tener la clase “Coche”, que sería el conjunto formado por todos los coches.



Por tanto, la clase es un concepto abstracto. Cuando decimos “piensa en un coche”, todos tenemos en mente la idea general de un coche, con sus puertas, ruedas, volante... Sin embargo, cuando decimos “mira ese coche que está aparcado al lado del árbol” ya se trata de un coche determinado, con una matrícula concreta, un color concreto... y eso ya sería un objeto de la clase “Coche”. Por tanto, la clase es la idea que define y agrupa a los objetos concretos.

Las clases van a ser nuevos tipos de datos, que llamaremos **tipos de dato referencia**, y son diferentes de los tipos básicos que estudiamos en el tema anterior. Los tipos básicos son siempre los mismos y nunca cambian. Por su parte, los tipos referencia varían, ya que en cualquier momento podemos añadir a nuestro programa librerías que contengan nuevos tipos referencia.

Es importante destacar que cuando comenzamos a hacer un programa, tenemos a nuestra disposición todos los tipos básicos, y los tipos referencia predefinidos que vienen con Java, como por ejemplo, el tipo String. Añadiendo librerías, podremos conseguir nuevos tipos referencia, y para saber cómo manejarlos, deberemos consultar sus instrucciones de uso.

Tipos de datos básicos	Tipos de datos referencia
byte	String
short	Perro
int	Balón
long	Robot
double	Dinosaurio
float	Polinomio
boolean	Quiniela
char	CuentaCorriente
	...

3.1.- Los objetos

La definición de objeto es bastante sencilla una vez comprendido el concepto de clase:

Un **objeto** es una **variable** cuyo tipo de dato es una clase

Por tanto, una clase es un tipo de dato que representa algo de la vida real y un objeto es una variable cuyo tipo de dato es una clase. Por ejemplo, si tenemos la clase Coche, dos objetos distintos de esa clase serían:

- El coche de matrícula GR-2724-F, que es un objeto de la clase Coche y tiene una marca concreta (Citroën), un color (blanco), etc...
- El coche de matrícula 7136-DKL es otro objeto diferente de la clase Coche. Este objeto tiene otra marca (Volkswagen), un color (azul oscuro), etc...



Matrícula: GR-2724-F
Marca: Citroën
Color: Blanco



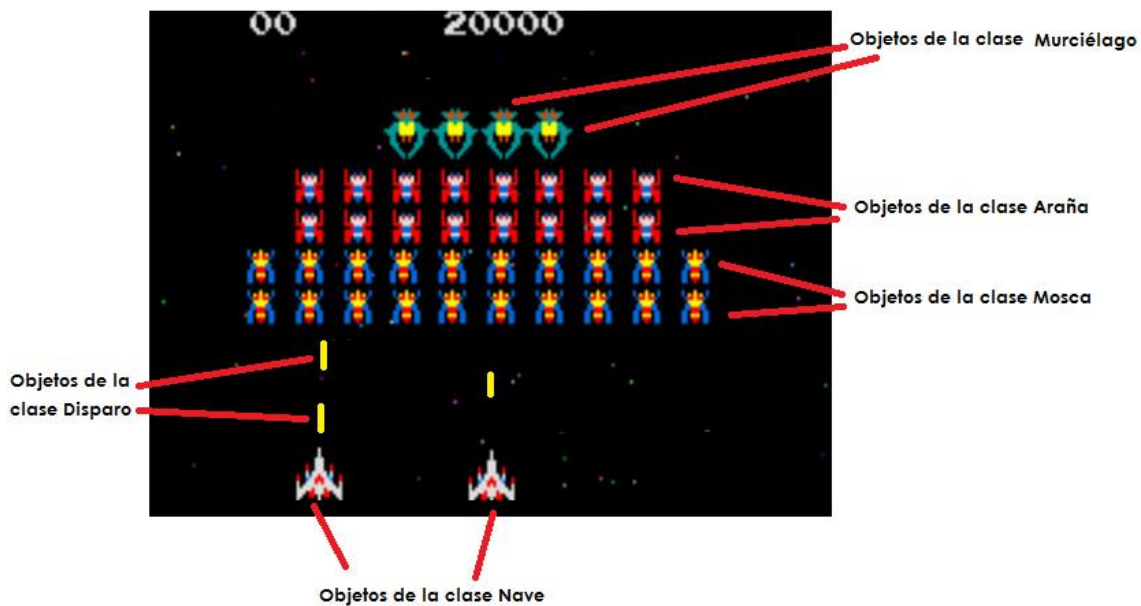
Matrícula: 7136-DKL
Marca: Volkswagen
Color: Azul oscuro

Observa como los dos coches tienen las mismas características (matrícula, marca, color), pero con valores diferentes que permiten distinguir un objeto de otro.

Los objetos también se llaman **instancias**, por lo que es frecuente que escuchemos frases como las siguientes, y todas significan lo mismo:

- El coche de matrícula GR-2724-F es un objeto de la clase Coche
- El coche de matrícula GR-2724-F es una instancia de la clase Coche
- El coche de matrícula GR-2724-F es una variable cuyo tipo de dato es Coche

Ejemplo: La programación orientada a objetos es genial para hacer videojuegos. Por ejemplo, un rápido vistazo a la pantalla del clásico “Galaga” nos permite descubrir cinco clases diferentes: Nave, Murciélago, Araña, Mosca y Disparo.



Como vemos en la imagen, durante el juego se están manejando varios objetos de cada una de las cinco clases del juego:

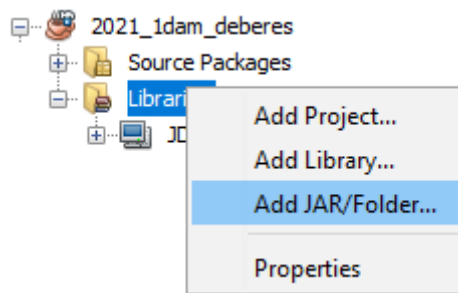
- Clase Nave: Hay dos objetos, que corresponden a las naves de los dos jugadores que están jugando simultáneamente. Cada uno estará guardado en una variable de tipo Nave. Estas variables pueden llamarse por ejemplo, "player1" y "player2".
- Clase Murciélago: Hay cuatro objetos diferentes.
- Clase Araña: Hay 16 objetos distribuidos en dos filas de ocho objetos cada una.
- Clase Mosca: Hay 20 objetos distribuidos en dos filas de diez objetos cada una.
- Clase Disparo: El jugador uno ha disparado dos veces, lo que ha creado dos objetos de la clase Disparo. El jugador dos ha disparado una sola vez.

3.3.- Utilización de una librería

Hemos visto que las clases se distribuyen en archivos llamados **librerías**, que han sido programados por programadores de clases. Entonces, si queremos hacer un programa que utilice una clase que está dentro de una librería, lo primero que tendremos que hacer es agregar el archivo de la librería al IDE que estemos manejando.

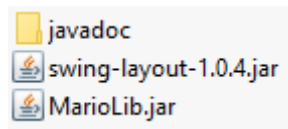
La forma de hacer este primer paso depende del IDE que utilicemos, y en el tema 1 de la asignatura de Entornos de Desarrollo aparece explicado con todo detalle para NetBeans y Eclipse. Allí podremos ver cómo agregar el archivo JAR de la librería a un proyecto concreto, y en caso de que la librería la utilicemos con mucha frecuencia, también viene explicado cómo se puede registrar la librería en el IDE, para que añadir la librería sea mucho más fácil aún y no nos dé problema cuando se copia un proyecto de un equipo a otro.

Para agregar una librería en NetBeans, seleccionaremos la carpeta "libraries" y con el botón derecho sobre ella pulsamos "Add Jar/Folder" y elegimos el archivo con la librería.



Una vez que se ha agregado la librería al IDE, el siguiente paso es importar el paquete donde se encuentra la clase que queremos utilizar. Como hemos dicho, un paquete no es más que una carpeta que hay dentro de la librería que sirve para agrupar clases de manera ordenada. La importación del paquete se hace en el código fuente utilizando la palabra **import** como vamos a ver a continuación.

Vamos a trabajar con la librería Mario Bros, que tenemos en la carpeta de librerías del tema. Esta librería está formada por dos archivos JAR y una carpeta llamada “javadoc”, donde se encuentran las instrucciones que describen las clases que hay en la librería y su uso.



Si entramos en la carpeta de la documentación y abrimos el archivo “index.html”, veremos que la librería tiene un paquete llamado “bpc.daw.mario” y en él se encuentran las clases Mario, Luigi, Planta, Seta, Cañón y Disparo.

Package **bpc.daw.mario** — NOMBRE DEL PAQUETE

Class Summary

Class	Description
Cañon	Crea un cañon capaz de disparar
Disparo	Esta clase representa los disparos de un cañon
Luigi	Esta clase representa un muñeco de Luigi
Mario	Esta clase representa un muñeco de Mario
Planta	Esta clase es una planta carnívora
Seta	Esta clase es un muñeco con forma de seta

CLASES
DEL PAQUETE
bpc.daw.mario

Esto significa que podemos hacer un programa y crear una variable cuyo tipo de dato sea Mario, o Luigi o Seta. Pero primero, tendremos que agregar los archivos jar de la librería al IDE (como ya vimos en la página anterior) y luego, en nuestro programa, importar el paquete bpc.asir.mario, que es donde están esas clases. Si no importamos el paquete, Java no reconocerá los tipos de datos de la librería, aunque hayamos agregado el jar al IDE.

Para importar el paquete, creamos un nuevo programa y al principio, justo antes de la línea “public class” escribimos:

`import bpc.daw.mario.*;`
importamos nombre del paquete importamos todas sus clases

Una vez escrita la orden anterior, todas las clases del paquete `bpc.asir.mario` estarán a disposición para nuestro programa. A partir de ese momento, Java reconocerá los tipos de dato Mario, Luigi, Seta, etc. Utilizar el `*` significa que se importan todas las clases que hay en el paquete `bpc.asir.mario`.

Si en lugar de importar todas las clases queremos importar solo una clase, lo que se hace es escribir su nombre en lugar del `*`, así:

`import bpc.daw.mario.Luigi;`
importamos nombre del paquete importamos solo la clase Luigi

Escribiendo esta última línea, podríamos utilizar en nuestro programa solamente la clase Luigi. Si intentásemos utilizar las clases Mario, Disparo o Planta nos saldría un error de compilación, por ser clases no importadas.

Una vez que tenemos importada la librería, podemos crear variables de estos nuevos tipos de datos. Una característica muy importante que tienen las variables de tipo referencia es que podemos dejarlas vacías² asignándoles la palabra **null**. Por tanto, podemos hacer esto:

```
1  import bpc.daw.mario.*;
2  public class Programa{
3      public static void main(String[] args){
4          Mario m = null;
5          Seta s = null;
6          Luigi l = null;
7      }
8  }
```

Este programa simplemente crea tres variables “m”, “s” y “l”, cuyos tipos de dato son Mario, Seta y Luigi, pero ahora mismo esas variables están vacías y no tienen ningún objeto dentro.

☞ **Curiosidad:** ¿Es posible utilizar una clase sin poner import? Pues sí, es posible. Bastará con escribir la clase mediante su “nombre completamente cualificado”, que consiste en ponerle como prefijo el nombre de su paquete. Mira este código, donde se hace lo mismo que el de antes, pero sin usar ningún import:

² Dejar vacío una variable de tipo básico es algo impensable, puesto que siempre deben contener un valor en la memoria ram.

```

1 public class Programa{
2     public static void main(String[] args){
3         bpc.daw.Mario m = null;
4         bpc.daw.Seta s = null;
5         bpc.daw.Luigi l = null;
6     }
7 }

```

Como puede verse, es mucho mejor usar import porque nos evita escribir un montón. El uso del nombre completamente cualificado solamente está justificado cuando se utilizan clases que se llaman igual y que se encuentran en paquetes diferentes.

4.- Métodos constructores

Una vez que hemos agregado la librería al IDE y hemos importado un paquete, ya estamos en condiciones de utilizar los objetos que contiene.

En el último código de ejemplo del punto anterior, creamos tres variables de tipo Mario, Seta y Luigi, pero esas variables se dejaron vacías (guardan **null**). Por tanto, en ellas no hay nada. No guardan ningún objeto al que podamos dar órdenes o hacer preguntas.

Por tanto, lo primero que habrá que hacer será dar vida a los objetos y guardarlos en sus variables correspondientes, que ahora mismo tenemos vacías.

Para crear objetos, utilizaremos los **métodos constructores**. Si observamos la documentación de la clase Mario podremos ver un apartado titulado “Constructor summary”:

Constructors	
Constructor and Description	
<code>Mario()</code>	Crea un muñeco de Mario en la esquina superior izquierda de la pantalla
<code>Mario(double r)</code>	Crea un muñeco de Mario a una distancia en píxeles medida desde la esquina superior izquierda de la pantalla
<code>Mario(int x, int y)</code>	Crea un muñeco de Mario en unas coordenadas de la pantalla
<code>Mario(Luigi l, int pix)</code>	Crea un muñeco de Mario al lado del muñeco de Luigi que se pasa como parámetro.

Vemos que la clase Mario tiene cuatro métodos constructores. Eso significa que nos proporciona cuatro formas distintas de crear un objeto Mario, y que podemos usar la que más nos convenga.

Si queremos crear un muñeco de Mario para que aparezca en una posición concreta de la pantalla, a partir de sus coordenadas (x,y), usamos el tercer constructor. Para ello escribiremos esto, donde antes teníamos Mario m = null;

```
Mario m = new Mario(120,90);
```

↖
↗
↖

objeto *clase* *parámetros que pide el constructor elegido*

La línea anterior hace que se cree en la memoria ram un objeto de la clase Mario que está situado en la posición (120,90) de la pantalla, y se guarda en la variable “m”. Si lo ejecutamos, podremos ver un muñeco como este:



Ahora “m” ya no está vacía. Tiene un objeto al que podemos dar órdenes y hacer preguntas, tal y como veremos en los apartados siguientes.

Podemos crear todos los muñecos que queramos, siempre que los guardemos en variables diferentes. Además, podemos usar cualquier constructor. Por ejemplo, si quisiéramos crear un muñeco de Mario en la esquina superior izquierda de la pantalla vemos en la documentación que el constructor que más nos interesa es el primero. Se usaría así:

```
Mario m = new Mario();
```

Observa que en este caso no se pone nada dentro de los paréntesis, porque así nos lo exigen las instrucciones de uso de ese constructor. Aún así, los paréntesis hay que ponerlos.

Supongamos ahora que queremos un muñeco de Mario que esté situado a 180 píxeles de la esquina de la pantalla, en diagonal. El constructor que mejor viene es el segundo, así:

```
Mario m = new Mario(180);
```

Por último, no debemos olvidar que los tipos referencia son tipos de datos, y que pueden formar parte de los métodos constructores. En el último constructor vemos que podemos crear un muñeco de Mario al lado de otro muñeco de Luigi que ya esté hecho. Para utilizarlo, comenzamos creando un muñeco de Luigi y lo guardamos en una variable llamada “luigi” (no confundir “luigi” con “Luigi”, que son palabras distintas). A continuación, creamos el muñeco de Mario con el cuarto método constructor. Como primer parámetro pasamos la variable “luigi” que contiene el muñeco ya creado de Luigi. El segundo parámetro sirve para indicar una cantidad de píxeles para separarlos (si se pone 0, la librería los separa automáticamente).

```
Luigi luigi = new Luigi(120,90);
Mario m = new Mario (luigi,0);
```

Si ejecutamos las líneas anteriores, veremos que el muñeco de Mario se ha creado justo a continuación del muñeco de Luigi:



Como puede verse, el programador de aplicaciones tiene libertad para elegir el constructor que mejor se adapte a lo que quiera hacer.

Ejercicio 2 : Consulta la documentación de la librería Mario Bros y haz un programa que cree los siguientes objetos en la pantalla:

- a) Dos muñecos de Mario, uno a 100 píxeles de distancia de la esquina superior izquierda de la pantalla y otro en las coordenadas (640,320)
- b) Una Seta en la esquina superior izquierda de la pantalla
- c) Un cañón en las coordenadas (100,320)
- d) Tres plantas en las coordenadas (400,500), (450,500) y (500,500)
- e) Un muñeco de Luigi situado a 200 píxeles de la esquina superior izquierda de la pantalla y uno de Mario situado a 50 píxeles horizontales de él.

5.- Métodos de instancia

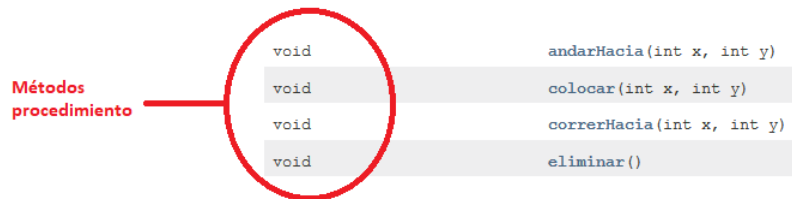
Una vez que un objeto ha sido creado mediante un constructor, es hora de darle órdenes y que haga cosas útiles para nuestro programa. Cada objeto va a tener un repertorio de órdenes diferentes llamadas **métodos de instancia** (en la práctica se les llama solamente “métodos”). Su nombre y función viene explicado en la documentación, en la sección titulada “method summary”.

Method Summary

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type		Method and Description	
void		andarHacia(int x, int y)	
void		colocar(int x, int y)	
void		correrHacia(int x, int y)	
void		eliminar()	
double		getDistanciaOrigen()	
int		getX()	
int		getY()	
void		girar()	
static void		main(java.lang.String[] args)	
void		pensar()	
void		saltar()	

Hay dos tipos de métodos:

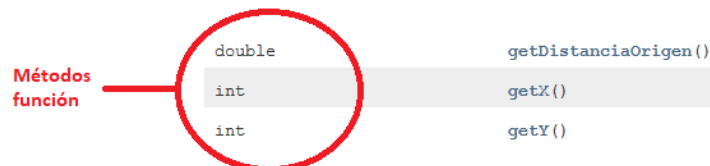
- **Métodos procedimiento:** Son órdenes que le damos a un objeto y este nos obedecerá. Los reconocemos porque en la primera columna de su documentación aparece la palabra **void**. Por ejemplo, en las instrucciones de la clase Mario podemos ver que “andarHacia”, “saltar”, “correrHacia”, etc son métodos procedimiento.



A red circle highlights the first column of the table, with a red line pointing to the label 'Métodos procedimiento'.

void	andarHacia(int x, int y)
void	colocar(int x, int y)
void	correrHacia(int x, int y)
void	eliminar()

- **Métodos función:** Son métodos que sirven para pedirle información al objeto. Es decir, el objeto nos responde con un dato, que **deberemos guardar en una nueva variable**³. Reconoceremos estos métodos porque en la primera columna encontramos el **tipo de dato** de la respuesta que nos da el objeto. Por ejemplo, en la clase Mario podemos encontrar el método “getDistanciaOrigen”, que es un método función y su respuesta nos da un dato de tipo double, que deberemos guardar en una nueva variable.



A red circle highlights the first column of the table, with a red line pointing to the label 'Métodos función'.

double	getDistanciaOrigen()
int	getX()
int	getY()

En los siguientes apartados vamos a aprender a utilizar métodos procedimiento y métodos función. Cuando tenemos un objeto y utilizamos un método, decimos que “*se llama a ese método*”.

4.1.- Uso de métodos procedimiento

Utilizar un método procedimiento es muy fácil. Basta escribir el nombre de la variable del objeto que va a recibir la orden, y después poner un punto (.), el nombre del método y encerrar entre paréntesis todos los datos que nos indique la documentación del método.

Por ejemplo, supongamos que queremos que nuestro muñeco de Mario ande hacia el punto de coordenadas (500,700). Si miramos la documentación de la clase Mario, veremos que tiene este método:

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type		Method and Description	
void		andarHacia(int x, int y)	

³ La **principal causa de suspensos** es usar un método función y no guardar la respuesta del objeto en una variable. De esta forma, la respuesta se pierde y es imposible continuar haciendo el programa.

El método “andarHacia” realiza justo lo que queremos hacer. Para darle dicha orden al muñeco que tenemos guardado en una variable llamada “m”, escribimos lo siguiente:

m.andarHacia(500,700);
↑ ↑ ↙
objeto método parámetros que pide el método

Por tanto, el programa completo quedaría así:

```
1 // El import es necesario para poder usar la clase Mario
2 import bpc.daw.mario.*;
3
4 public class Programa{
5     public static void main(String[] args){
6         // Creo un muñeco de Mario en las coordendas (120,90)
7         Mario m = new Mario(120,90);
8         // Ordeno al muñeco que ande hacia la posición (500,700)
9         m.andarHacia(500,700);
10    }
11 }
```

Si se ejecuta, se puede ver cómo el muñeco obedece y se pone a andar hacia el punto que le indiquemos.

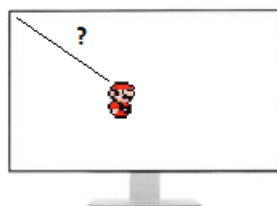
Ejercicio 3 : Haz un programa en el que se vea un muñeco de Mario, un Luigi, una seta, tres plantas y un cañón situados donde quieras. Realiza con los muñecos las siguientes acciones:

- a) La seta debe moverse hacia la esquina superior izquierda de la pantalla.
- b) El muñeco de Mario debe pegar un salto.
- c) El cañón debe disparar al muñeco de Luigi.
- d) Las plantas tienen que estar comiendo.

5.2.- Uso de métodos función

Como ya hemos dicho, un método función es una solicitud de información a un objeto. Lo importante es que cuando tenemos un método función, el objeto nos responde con algo que deberemos guardar en una variable. La utilización de un método función se hace igual a la de un método procedimiento, pero **guardando el resultado del método en una variable** del tipo de dato que indiquen las instrucciones del método.

Por ejemplo, supongamos que creamos un muñeco de Mario en una variable llamada “m” y queremos preguntarle a qué distancia se encuentra de la esquina superior izquierda de la pantalla.



```
double      getDistanciaOrigen()
```

Para llamar al método `getDistanciaOrigen` sobre el objeto “m” y guardar su respuesta en una variable llamada “distancia” escribiremos esto:

El programa completo sería así:

Como ya hemos dicho varias veces, es muy importante recoger la respuesta en una nueva variable. Observa que la variable “distancia” es necesaria para poder escribir la línea 12 del programa. Es posible llamar al método sin recoger la respuesta, pero entonces no podríamos continuar haciendo el programa porque la respuesta se habría perdido y ya no la podríamos mostrar por pantalla. Es decir, **lo siguiente estaría mal:**

⁴ Recuerda que “llamar a un método” significa lo mismo que “utilizar el método”.

También es posible mostrar la respuesta del objeto directamente en la pantalla, así:

```
1  import bpc.daw.mario.*;
2
3  public class Programa{
4      public static void main(String[] args){
5          Mario m = new Mario(120,90);
6          // Pregunto la distancia y la muestro por pantalla
7          System.out.println("La distancia es: "+m.getDistanciaOrigen());
8      }
9  }
```

Haciendo esto, en la pantalla se verá el valor double que devuelve el método `getDistanciaOrigen`. Pero si el programa continúa y más tarde necesita otra vez la distancia, deberemos volver a preguntársela al objeto, ya que en el código anterior se ha mostrado por pantalla, pero no se ha guardado en ninguna variable.

Ejercicio 4 : Haz un programa en el que se cree un muñeco de Luigi situado a una distancia en diagonal, de 500 píxeles a partir de la esquina superior izquierda de la pantalla, y un muñeco de Mario a su lado. El programa deberá mostrar por pantalla las coordenadas (x,y) en la que se encuentran los dos muñecos.

6.- Documentación de las clases

Durante todo el tema estamos viendo la importancia que tiene la lectura y comprensión de la documentación que realiza el programador de clases. Si no se tienen claros los métodos de los objetos y como usarlos, el programador de aplicaciones no podrá hacer programas con ellos.

Existen dos formatos que el programador de clases puede utilizar para dar a conocer la documentación de la clase, que vamos a ver en los siguientes apartados:

6.1.- Javadoc

Es el formato que hemos visto en los ejemplos de la librería MarioLib. Se trata de una página web en la que aparece toda la información de las clases, incluyendo una descripción, sus constructores y el resto de métodos.

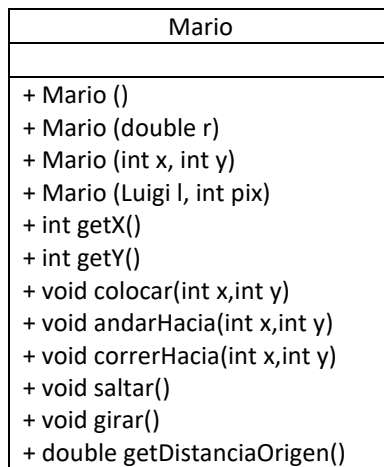
☞ **Curiosidad:** *¿Quién hace el javadoc de una librería?* Como es lógico, lo hace el programador de clases que fabrica la librería. Sin embargo, no lo hace escribiendo a mano todo el código HTML. El Javadoc lo genera automáticamente el IDE a partir de los comentarios que se escriben en el código fuente de la clase. Todo esto se estudia en la asignatura de Entornos de Desarrollo.

6.2.- Diagrama de clases

Es una forma estándar de representar las clases de manera gráfica. Cada clase de la librería se representa mediante un rectángulo que está dividido en tres zonas:

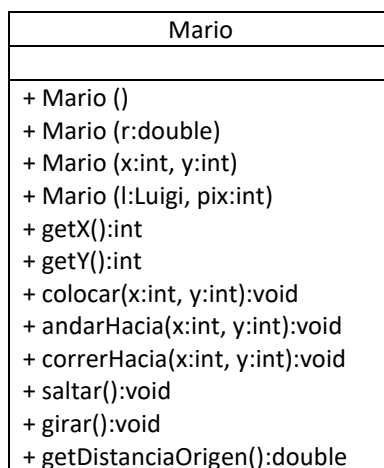
- Zona superior: Aparece el nombre de la clase
- Zona intermedia: De momento la veremos en blanco. Es usada por el programador de la clase.
- Zona inferior: Aparecen los métodos de la clase.

En el lenguaje Java el diagrama de clases de la clase Mario se representaría así:



Como se puede observar, los métodos constructores se identifican porque su nombre es igual al nombre de la clase y no tienen nada a su izquierda (ni void ni tipo de dato). Los demás métodos siguen un formato similar al que hemos visto en el Javadoc.

Esta forma de escribir el diagrama de clases es específica del lenguaje Java, pero no es aplicable al resto de lenguajes. Existe un estándar denominado **UML** que estandariza todos los diagramas utilizados en el proceso de construcción de un software orientado a objetos. En UML, la forma de indicar los parámetros y el retorno cambia, siendo así:



El diagrama de clases del proyecto consiste en una hoja con todas las representaciones de las clases que intervienen, y también se indican las **relaciones** que hay entre las clases. UML y el diagrama de clases en profundidad se estudian en la asignatura de Entornos de Desarrollo.

En cualquier caso, acompañando al diagrama de clases siempre aparece una explicación textual de todos los métodos que tiene la clase. Por ejemplo, para la clase Mario la especificación podría ser esta:

ESPECIFICACIÓN DE LA CLASE MARIO

Descripción: Esta clase representa el muñeco de Mario y permite realizar algunas acciones sencillas con él.

Paquete: bpc.daw.mario

Observaciones: Todo lo que aquí se va a explicar sirve igual para la clase Luigi.

Diagrama de clase:

Mario
+ Mario () + Mario (double r) + Mario (int x, int y) + Mario (Luigi l, int pix) + int getX() + int getY() + void colocar(int x,int y) + void andarHacia(int x,int y) + void correrHacia(int x,int y)

Métodos:

- El constructor sin parámetros crea un muñeco de Mario en la esquina superior izquierda de la pantalla
- El segundo constructor crea un muñeco de Mario situado en diagonal, a la distancia indicada de la esquina superior izquierda de la pantalla.
- El tercer constructor crea un objeto Mario en las coordenadas (x,y) recibidas como parámetro.
- El cuarto constructor crea un muñeco de Mario al lado del muñeco de Luigi que se indica como parámetro. El segundo parámetro es la distancia horizontal a la que se encuentran ambos muñecos. Si se pone 0, esa distancia se ajusta automáticamente
- El método getX devuelve el valor de la coordenada del eje X donde se encuentra el muñeco en un momento dado. El método getY realiza lo mismo pero con el eje Y.
- El método colocar traslada inmediatamente al muñeco a las coordenadas (x,y) pasadas como parámetro.
- El método andarHacia pone a caminar al muñeco hacia las coordenadas (x,y) pasadas como parámetro.
- El método correrHacia hace que el muñeco se mueva hacia las coordenadas (x,y) pasadas como parámetro de forma más rápida.

Ejercicio 5 : Realiza un programa con la librería de Mario Bros en el que haya dos cañones. Haz que el primero dispare dos veces y el segundo tres. Pregunta a los dos cañones cuántas veces han disparado y muestra dichos valores por pantalla.

Ejercicio 6 : Crea un programa que pida por teclado dos coordenadas y cree un muñeco de Mario en ellas.

Ejercicio 7 : Haz un proyecto nuevo y añade la librería “ObjetosSencillos.jar”. En dicha librería encontrarás la clase **DepósitoAgua**. Haz un programa en el que haya un depósito de agua con capacidad máxima 50 litros y capacidad inicial 15 litros. Dibújalo en la pantalla y calcula el tanto por ciento de ocupación del depósito.

Ejercicio 8 : Haz un programa en el que haya dos depósitos de agua de capacidad máxima 20 litros y con capacidades iniciales 15 y 5 litros respectivamente. Retira del primero cinco litros de agua y añádelos al segundo. Dibuja los depósitos antes y después de la operación.

Ejercicio 9 : En la librería ObjetosSencillos.jar se encuentra la clase **Caja**. Haz un programa que cree una caja que contenga el mensaje “Bienvenidos al instituto”. Consultar el mensaje con la caja cerrada y mostrar el resultado. Abrir la caja y consultar el mensaje. Mostrar el resultado.

Ejercicio 10 : Haz otro programa que cree una caja con un mensaje cualquiera. Abrir la caja, consultar el mensaje y mostrarlo por pantalla. Sustituir el mensaje por otro. Volver a consultarlo y mostrar el resultado.

Ejercicio 11 : Haz otro programa que cree dos cajas, cada una con un mensaje diferente. Hacer un programa que intercambie los mensajes de las dos cajas.

Ejercicio 12 : En la librería ObjetosSencillos.jar se encuentra la clase **MarcadorBaloncesto**. Haz un programa en el que se cree un partido de baloncesto para el partido que van a jugar el Estudiantes y el CB Granada. Registra las siguientes canastas y muestra el nombre de los equipos, los puntos que tiene cada uno, el nombre del equipo que va ganando y el que va perdiendo.

E ->2, CB -> 3, E->2, CB->2, CB->3, E->1, E->1, CB->2

Ejercicio 13 : Consulta el pdf de la “Librería Estándar de Java” y busca la clase **String**. Haz un programa que pregunte tu nombre por teclado, y el ordenador lo muestre en mayúsculas, minúsculas y diga cuántos caracteres tiene.

```
Escribe tu nombre:
Juan Diego
Tu nombre en mayúsculas: JUAN DIEGO
Tu nombre en minúsculas: juan diego
Total de caracteres de tu nombre: 10
```

Ejercicio 14 : Consulta la documentación de la clase String para hacer un programa que pregunte una frase al usuario y se muestre la letra que hay justo en la mitad de la frase.

Ejercicio 15 : Haz un programa en el que haya una variable String llamada "clave" con el texto que tú quieras. A continuación, el programa pedirá que introduzcas una contraseña por teclado. El ordenador mostrará si la variable "clave" coincide con la contraseña introducida.

Ejercicio 16 : Haz un programa que pregunte al usuario por teclado dos frases. El programa deberá mostrar por pantalla si la segunda frase está incluida dentro de la primera frase.

```
Introduzca el texto 1
Nos vamos a la playa
Introduzca el texto 2
playa
¿El texto 2 está incluido en el texto 1? true
```

Ejercicio 17 : Como ya sabes, una dirección de correo tiene el formato nombre@dominio. Realiza un programa que pregunte al usuario su nombre y después su dominio. En caso de que el dominio termine en .com o .es, el programa el programa formará una nueva variable llamada "correo" a partir de ellos y la mostrará por pantalla. En caso contrario, mostrará un mensaje de error "El dominio es incorrecto".

```
¿Cuál es tu nombre?
juandiego
¿Cuál es tu dominio?
gmail.com
Tu correo es: juandiego@gmail.com
```

Ejercicio 18 : Haz un programa que pregunte al usuario por su dirección de correo y la descomponga en su nombre de usuario y su dominio.

```
¿Cuál es tu correo electrónico?
juandiego@gmail.com
- Tu nombre de usuario es: juandiego
- Tu dominio es: gmail.com
```

Ejercicio 19 : Consulta el pdf de la librería "java io" y encuentra la clase **File**. Realiza un programa que pregunte al usuario por teclado la ruta de un archivo. El programa deberá decir cuántos bytes y cuántos megabytes ocupa en el disco duro.

Ejercicio 20 : Realiza un programa que pregunte al usuario por la ruta de un archivo del disco duro. Si dicho archivo existe y es directorio, el programa dirá "La ruta introducida corresponde a un directorio" y terminará sin hacer nada más. En caso de que sea un archivo, el programa mostrará "*La ruta introducida es un archivo de tamaño ... bytes. ¿Desea borrarlo (si/no)?*". Si el usuario introduce la palabra "si", el archivo se borrará y el ordenador mostrará si se ha borrado correctamente o se ha producido un error al borrar. En caso de que el usuario escriba cualquier otra cosa el programa responderá "Borrado cancelado" y finalizará.

Ejercicio 21 : Realiza un programa que pida por teclado la ruta de un directorio. En caso de que la ruta introducida no sea un directorio, el programa mostrará un mensaje de error. En caso contrario, el programa mostrará el número total de archivos y carpetas que contiene.

7.- Excepciones

Algunos métodos que podemos encontrarnos son “peligrosos”, es decir, su llamada puede provocar una **excepción**, o error cuando el programa está funcionando. Por ejemplo, supongamos que tenemos una clase Impresora y un método imprimir(). Podría ocurrir que al poner en marcha el programa, el método imprimir funcione correctamente, pero si por ejemplo no hay tinta, dicho método fallará y si no hacemos nada, dará un error que interrumpa el funcionamiento del programa, mostrando una ventana como esta:



Esto significa que hay métodos que no sabemos si van a fallar o no, porque eso va a depender de las condiciones del sistema cuando se ejecuta el programa. Algunos ejemplos de más situaciones en las que puede producirse una excepción son:

- Un usuario escribe una palabra cuando el ordenador está esperando un número
- Un programa intenta acceder a un fichero que no existe o no tiene permisos de lectura
- Un programa intenta realizar una división por cero.
- Un programa intenta llamar a un método sobre una variable que guarda null.

Si no hacemos nada, cuando se produzcan esas situaciones el programa terminará de forma anormal, y el usuario perderá todo lo que haya hecho, siendo imposible recuperar la situación. Como se puede imaginar, esto denota falta de profesionalidad por parte de los programadores que hicieron la aplicación.

En Java es posible anticiparnos a los fallos que puedan lanzar los métodos, para poder continuar con normalidad la ejecución del programa en caso de que se produzcan.

7.1.- Reconocimiento de métodos “peligrosos”

¿Cómo podemos saber si un método es “seguro” o “peligroso”? en Java los métodos que pueden lanzar excepciones aparecen marcados en sus instrucciones con la palabra “throws” y a continuación, viene el nombre del tipo de excepción que pueden lanzar.

Vamos a trabajar con la librería AscensorLib de la carpeta de recursos del tema. Si abrimos su javadoc veremos que tiene una clase Ascensor a la que podemos añadir personas y mover el ascensor entre sus plantas. En la vista general de la librería no observamos nada raro:

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
void	<code>añadir(Persona p)</code> Añade una persona al ascensor.	
int	<code>getNumeroPersonas()</code> Devuelve el número de personas que hay en el ascensor	
double	<code>getPesoActual()</code> Devuelve el peso actual de todas las personas que hay en el ascensor	
int	<code>getPlantaActual()</code> Devuelve el número de planta actual en el que está el ascensor	
void	<code>mover(int numPlanta)</code> Mueve el ascensor a la planta indicada.	
void	<code>retirar(Persona p)</code> Retira una persona del ascensor.	
java.lang.String	<code>toString()</code> Devuelve información sobre el ascensor	

Pero si bajamos más, o pulsamos el enlace que nos lleva a los detalles del método “añadir”, veremos que allí aparece “throws PesoMaximoException”

añadir

```
public void añadir(Persona p)
    throws PesoMaximoException
```

Añade una persona al ascensor. Si se excede el peso máximo, se lanza una `PesoMaximoException`

Throws:
`PesoMaximoException`

El método puede lanzar una excepción de tipo `PesoMaximoException`

¿Qué significa eso? Pues que ese método habrá veces que funcione correctamente, pero en otros casos puede lanzar una excepción. Si leemos las instrucciones nos daremos cuenta de que en este método la excepción se lanza cuando queremos añadir una persona al ascensor y se desborda el peso. Vemos que el nombre de la excepción que se lanza se llama `PesoMaximoException`, y es un nombre que ha puesto el programador de la librería.

Si observamos la librería, veremos que el método mover también es “peligroso”, porque puede lanzar una `PlantaIncorrectaException` en caso de que queramos llevar al ascensor a una planta que no exista.

7.2.- El bloque try-catch

El bloque try-catch es un mecanismo que protege la llamada a un método “peligroso”, para que el programa pueda reaccionar adecuadamente si se produce una excepción. Se trata de un bloque que encierra el trozo de código peligroso, y proporciona un bloque alternativo que se ejecutará en caso de que el trozo peligroso lance una excepción. Por supuesto, este bloque alternativo no se ejecutará si el trozo peligroso no produce ninguna excepción.

Por ejemplo, supongamos que creamos un objeto Ascensor que pueda moverse en un rango de plantas entre -1 y 5. Sería así:

```
3 public class Programa{
4     public static void main(String[] args){
5         // creo un ascensor que se mueve entre las plantas -1 y 5
6         Ascensor a = new Ascensor(-1,5);
7     }
8 }
```

Ahora queremos moverlo, por ejemplo a la planta 3. El método que hace esto es el método “mover”, que hemos visto antes que está marcado como “peligroso” en la documentación de la clase. Por este motivo, no podemos llamarlo como hemos hecho hasta ahora, sino que tenemos que encerrarlo en un bloque try-catch, así:

```
3 public class Programa{
4     public static void main(String[] args){
5         // creo un ascensor que se mueve entre las plantas -1 y 5
6         Ascensor a = new Ascensor(-1,5,500);
7         /* Muevo el ascensor a la planta 3. Como es una acción
8            que puede fallar, la protejo encerrándola en try-catch */
9         try{
10            a.mover(3);
11        }catch(Exception p){
12            // Esta línea se muestra si se produce una excepción cualquiera
13            System.out.println("Excepción al mover el ascensor");
14        }
15        System.out.println("Programa finalizado");
16    }
17 }
```

De esta forma, si el método mover funciona correctamente (como ocurre si la planta destino es la 3), el programa funciona bien y termina. Si ponemos un número de planta incorrecto, el método mover lanzará una excepción y el programa entrará en el bloque catch, donde se realizarán las acciones que allí se indican. Al terminar el catch, el programa prosigue su ejecución normal.

```
public class Programa{
    public static void main(String[] args){
        // creo un ascensor que se mueve entre las plantas -1 y 5
        Ascensor a = new Ascensor(-1,5);
        /* Muevo el ascensor a la planta 3. Como es una acción
           que puede fallar, la protejo encerrándola en try-catch */
        try{
            a.mover(3);
        }catch(Exception p){
            // Esta línea se muestra si se produce una excepción cualquiera
            System.out.println("Excepción al mover el ascensor");
        }
        System.out.println("Programa finalizado");
    }
}
```

si no hay error
después de mover
se termina el programa

si hay error,
el programa ejecuta
el bloque catch y luego
prosigue con normalidad

Por seguridad, Java obliga a encerrar en un bloque try-catch las excepciones que aparecen marcadas con la palabra “throws” en las declaraciones de los métodos. Por este motivo, llamar a un método “peligroso” sin try-catch dará un error de compilación.

Cuando se utiliza un bloque try-catch es **muy importante** no crear variables dentro de él, ya que dichas variables “mueren” al finalizar dicho bloque de llaves, y ya no las podríamos

utilizar después⁵. Por ejemplo, el siguiente programa daría un error de compilación porque intenta acceder a la variable “dato” fuera del try donde ha sido creada:

```
try{
    int dato=3;
    a.mover(dato);
}catch(Exception e){
    System.out.println(e.getMessage());
}
System.out.println("He movido el ascensor a la planta "+dato);
```



La variable "dato" ya no existe en este punto del código

Continuamos analizando el bloque try-catch. Cuando en el catch ponemos la palabra “Exception” lo que hacemos es capturar todas las posibles excepciones que puedan producirse en el interior del try. Pero también podemos escribir varios bloques catch diferenciando los tipos de excepciones, para así proporcionar respuestas distintas según el error producido.

Por ejemplo, el siguiente programa crea tres personas, las introduce en el ascensor y luego las lleva a la última planta.

```
Ascensor a = new Ascensor(-1,5,500);
Persona p1 = new Persona("Ana",60);
Persona p2 = new Persona("Carlos",100);
Persona p3 = new Persona("María",55);
try{
    a.añadir(p1);
    a.añadir(p2);
    a.añadir(p3);
    a.mover(5);
}catch(Exception p){
    System.out.println("Excepción producida por el ascensor");
}
System.out.println("Programa finalizado");
```

En este programa, si se produce una excepción, la respuesta será la misma sea cual sea la causa, por lo que el usuario tendría poca información sobre lo que ha pasado. Vamos a reescribirlo usando dos bloques catch en los que indicamos los tipos de excepciones que se pueden producir en el try:

```
Ascensor a = new Ascensor(-1,5,500);
Persona p1 = new Persona("Ana",60);
Persona p2 = new Persona("Carlos",100);
Persona p3 = new Persona("María",55);
try{
    a.añadir(p1);
    a.añadir(p2);
    a.añadir(p3);
    a.mover(5);
}catch(PesoMaximoException e){
    System.out.println("Se ha superado el peso máximo permitido");
}catch(PlantaIncorrectaException e){
    System.out.println("El número de planta indicado es incorrecto");
}
System.out.println("Programa finalizado");
```

⁵ En muchos lenguajes programación (como Java) las variables solo existen dentro del bloque de llaves donde se crean, y son borradas de la memoria al finalizar dicho bloque.

Ahora, en caso de que se produzca una excepción en el try, el programa se dirigirá al catch que captura el tipo de excepción indicado y ejecutará sus líneas de código.

Por último, vamos a centrarnos en la variable “e” que estamos poniendo dentro del catch. Esa variable es un objeto que representa a la excepción que se ha producido, y podemos utilizarla solamente en el bloque de llaves definido por su catch. Esa variable tiene un método llamado **getMessage()** que devuelve un String con un mensaje del error. Usando este método podremos ahorrarnos escribir explicaciones y utilizar la explicación que haya puesto el programador de la librería. Sería así:

```
try{
    a.añadir(p1);
    a.añadir(p2);
    a.añadir(p3);
    a.mover(5);
}catch(Exception e){
    System.out.println(e.getMessage());
}
```

En este programa, si se produce una excepción, el código entra en el catch y se muestra el mensaje que haya programado el programador de la librería cuando se produce la excepción. En este caso, la librería es inteligente y dicho mensaje mostrará si la excepción se debe a superar el peso máximo o a mover a una planta incorrecta. En otras librerías no habrá tanta suerte y el mensaje será poco indicativo.

Ejercicio 22 : En la librería *ObjetosSencillos.jar* se encuentra la clase **TarjetaCrédito**. Haz un programa que cree una tarjeta de crédito con contraseña 1111 y saldo 5000€. Mostrar la información de la cuenta por pantalla. Sacar 2000 € y volver a mostrar la información.

Ejercicio 23 : Haz un programa que cree una tarjeta de crédito con contraseña 2222 y saldo 1000€. Mostrar la información de la cuenta por pantalla. Ingresar 100 € y volver a mostrar la información. Retirar 2800 € de ella y volver a mostrar la información.

Ejercicio 24 : Consulta el pdf de la librería “java io” y busca la clase **PrintWriter**. Realiza un programa que pregunte por teclado al usuario dos frases. El ordenador creará un archivo llamado “d:/frases.txt” (asegúrate de tener permisos de escritura, si no, pon otra carpeta) y escribirá las dos frases en él.

Ejercicio 25 : Consulta el pdf de la “Librería Estándar de Java” y busca la clase **Scanner**. Realiza un programa que abra el archivo “d:/frase.txt” que has creado en el ejercicio anterior y nos muestre por pantalla las dos frases que hay guardadas en él.

Ejercicio 26 : Consulta el pdf de la “Librería Estándar de Java” y busca la clase **Thread**. Realiza un programa que pregunte al usuario “¿cuántos segundos quiere esperar?”. El usuario escribirá un número entero y el programa hará una pausa de dicha cantidad de segundos. Pasado ese tiempo se mostrará el mensaje “Programa finalizado”.

Ejercicio 27 : Consulta el pdf de la librería “java net” y busca la clase **InetAddress**. Realiza un programa que pregunte por teclado al usuario una dirección IP y un número llamado timeout. El ordenador nos indicará si puede hacer ping a la IP indicada, esperando el tiempo que dice la variable timeout.

```
Escriba una dirección IP
192.168.1.1
Escriba el tiempo de timeout (en milisegundos)
1000
Haciendo ping a 192.168.1.1 ... true
```

Ejercicio 28 : Usa la clase **InetAddress** para hacer un programa que pregunte por un nombre de equipo de la red y nos muestre su dirección IP.

```
Introduzca un nombre de equipo:
www.google.es
La IP de www.google.es es 172.217.168.163
```

7.3.- Tipos de excepciones

Como hemos visto, Java nos obliga a encerrar en un bloque try-catch todas las excepciones que aparecen con la palabra “throws” en la línea de definición del método. Observa que un poco más abajo también aparece “throws”, pero a ese no le vamos a hacer caso. Para saber si Java nos obliga a capturar una excepción con un try-catch, solo debemos fijarnos en el primer “throws”, el que hay más arriba en la línea donde se define el método.

añadir

```
public void añadir(Persona p)
    throws PesoMaximoException
```

Añade una persona al ascensor. Si se excede el peso máximo, se lanza una `PesoMaximoException`

Throws:

`PesoMaximoException`

El método
puede lanzar
una excepción
de tipo
`PesoMaximoException`

A estas excepciones se les llama **CheckedExceptions**, y como ya hemos dicho, es obligatorio capturarlas en un try-catch para no obtener un error de compilación.

Sin embargo, hay otro tipo de excepciones que **no** hay que capturarlas, y se llaman **RuntimeExceptions**. Estas excepciones se deben a errores nuestros que cometemos mientras programamos, y por tanto, debemos solucionarlos antes de seguir adelante.

Vamos a ver un ejemplo de `RuntimeException`. Supongamos que creamos una variable de tipo Mario y la dejamos vacía, tal y como hicimos al principio del tema:

```
3 public class Programa{
4     public static void main(String[] args){
5         Mario m = null;
6     }
7 }
```

A continuación, vamos a llamar al método `saltar()` sobre esa variable. Observa el código y la pregunta que nos hacemos en el comentario, porque la variable “m”, que debería

guardar un objeto Mario creado con new, en realidad no guarda nada y vale null. ¿Qué sucederá en este caso? Estamos llamando a un método sobre una variable que no guarda nada

```
3 public class Programa{
4     public static void main(String[] args){
5         Mario m = null;
6         m.saltar(); // ¿Puede saltar si "m" no tiene nada dentro?
7     }
8 }
```

Si pulsamos el botón de compilar, veremos que compila sin ningún problema, pero al ejecutarlo, se produce el error más temido en todos los lenguajes de programación⁶: el **NullPointerException**. Este error sale cuando tenemos un objeto que está vacío y llamamos a un método suyo. Es decir, la excepción salta al darle una orden a un objeto que guarda null.

```
Exception in thread "main" java.lang.NullPointerException
|   at bpc.daw.mario.Programa.main(Programa.java:19)
C:\Users\Juan Diego\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

¿Por qué el compilador no se ha dado cuenta de este error? Cuando escribimos código Java, el compilador comprueba que está bien escrito y que no tiene errores de escritura, pero no conoce lo que hay dentro de las variables. Por tanto, no sabe que “m” está vacío. Eso hace que el error se produzca cuando ponemos en marcha el programa.

La NullPointerException es un ejemplo de RuntimeException. Son excepciones que se producen por culpa nuestra cuando programamos. Por tanto, no debemos capturarlas, sino corregirlas. Otras RuntimeException que se producen fácilmente son:

- **ArrayIndexOutOfBoundsException**: Por ejemplo, tenemos la lista {"lunes", "martes"} y queremos acceder al dato de la posición 20 de esa lista (es una posición inexistente).
- **NumberFormatException**: Por ejemplo, tenemos un String que guarda “Hola” y lo queremos convertir en el tipo básico int con el método Integer.parseInt.
- **ArithmeticException**: Por ejemplo, hacemos una división de 50 entre 0.

Ejercicio 29 : Realiza las siguientes acciones en un programa y escribe si lanzan o no una excepción. En caso afirmativo, escribe el nombre de la excepción que se lanza.

- a) Por teclado se pide que el usuario escriba un número y el usuario escribe un texto
- b) Un programa necesita recuperar dos argumentos de la línea de comandos y no se pasa ninguno
- c) Creamos una caja vacía. Recuperamos el mensaje que tiene y mostramos por pantalla su longitud.

Ejercicio 30 : Busca en Internet cuándo se lanzan las siguientes excepciones, e indica si son de tipo RuntimeException o CheckedException.

- a) SQLException
- b) IllegalArgumentException
- c) IllegalStateException

⁶ En este caso solucionar el NullPointerException es muy sencillo, pero cuando el programa se hace grande la única forma posible es usar la característica de **depuración** que incorpora el IDE.

8.- Herencia

Uno de los conceptos fundamentales de la programación orientada a objetos es el de **herencia**, que refleja la relación “es un” que existe entre los objetos de la vida real. Por ejemplo, un teléfono móvil “es un” teléfono. Un empleado “es una” persona. Un loro “es un” pájaro, que a su vez “es un” animal...

Decimos que hay **herencia** entre dos clases A y B cuando “**A es un B**”

Por ejemplo, hay herencia entre la clase Loro y la clase Pájaro, o entre la clase Silla y la clase Mueble. Sin embargo, no hay herencia entre la clase Martillo y la clase Hospital.

Cuando hay herencia entre A y B se suele decir que “*A es una clase hija de B*” o que “*B es la clase padre de A*”. También hay personas que les gusta decir que “*A es una subclase de B*” o que “*B es una superclase de A*”.

¿Por qué es importante la herencia en la programación? Pues porque cuando hay herencia, **la clase hija recibe automáticamente los métodos de su clase padre**. Esto es muy importante, porque la clase hija recibe un montón de métodos ya programados, facilitando así la reutilización de código. Otra ventaja es que se puede usar la clase hija en cualquier lugar donde sea necesario un objeto de la clase padre.

Los únicos métodos que no se reciben de la clase padre son los constructores, así que la clase hija tendrá que programar sus propios métodos constructores.

Para saber si existe herencia entre dos clases, debemos mirar la documentación. Al principio del javadoc de una clase podremos ver el nombre de su clase padre al lado de la palabra “extends”.

Por ejemplo, si miramos la librería MusicaLib de la carpeta de recursos del tema, veremos que tenemos dos clases llamadas Instrumento y Guitarra. ¿Existirá herencia entre ellas? Desde luego tendría sentido, porque una Guitarra es un Instrumento. Para comprobarlo, abrimos el javadoc de la clase Guitarra y encontramos esto:

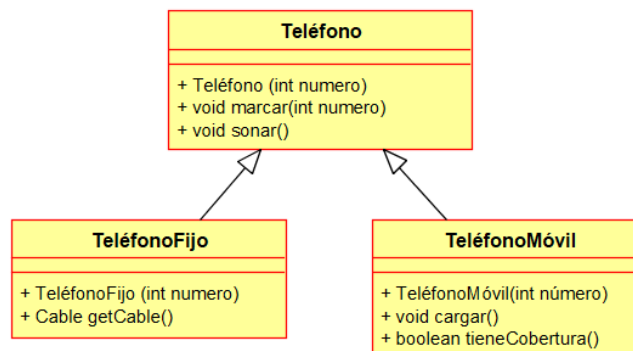
```
Class Guitarra

java.lang.Object
  bpc.daw.musica.Instrumento
    bpc.daw.musica.Guitarra

public class Guitarra
  extends Instrumento CLASE PADRE
```

Aquí podemos ver que la clase padre de Guitarra es la clase Instrumento, de manera que Guitarra heredará todos los métodos que hayan sido programados en Instrumento.

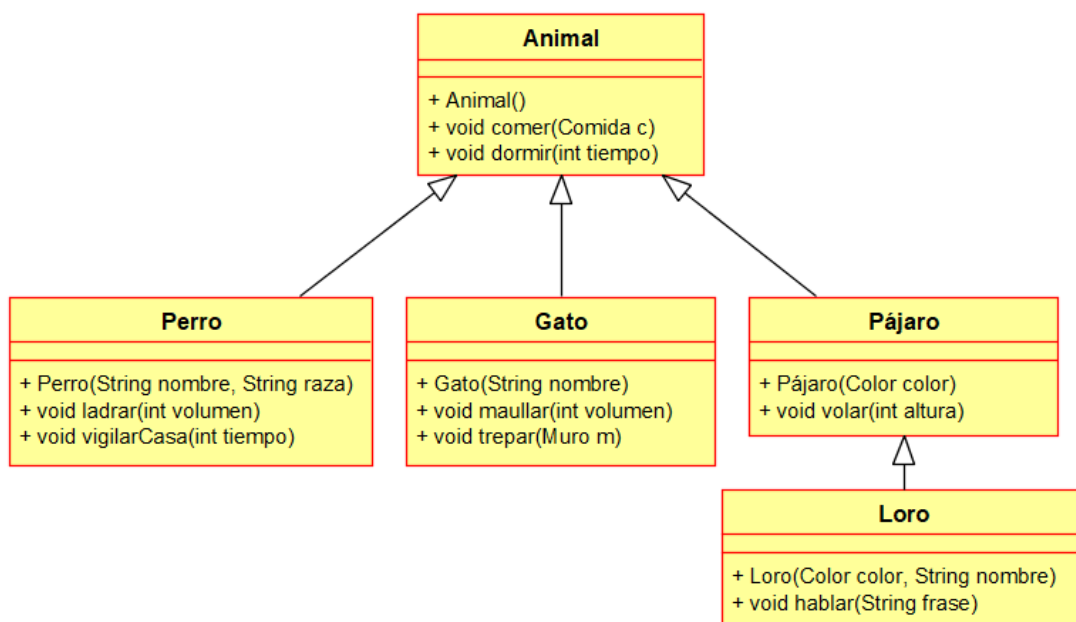
Cuando en lugar de un javadoc tenemos un diagrama de clases, la herencia se representa mediante una flecha hueca que va desde la clase hija hacia la clase padre, como por ejemplo vemos en este diagrama:



En esta situación las clases TeléfonoFijo y TeléfonoMóvil son dos clases hijas de la clase Teléfono. Eso significa que los objetos de la clase TeléfonoMóvil poseen dos métodos llamados MarcarNúmero y Sonar, que heredan de la clase padre. Además, cada clase hija tendrá sus propios métodos específicos, como por ejemplo el método cargar de la clase TeléfonoMóvil.

No hay límite para el número de clases de las que puede heredarse. Una clase puede heredar de otra que a su vez sea hija de otra y así sucesivamente. El diagrama de clases resultante se denomina “**árbol de herencia**”. Sin embargo, Java es un lenguaje en el que solo se puede dar la **herencia simple**⁷. Eso significa que una clase solo puede tener una única clase padre.

Aquí podemos ver un ejemplo de un árbol de herencia muy sencillo.



⁷ En realidad todos los lenguajes orientados a objetos actuales funcionan de esta forma, y solo C++ proporciona mecanismos para hacer herencia múltiple (una clase tiene más de una clase padre).

En este diagrama podemos ver como todos los animales heredan los métodos comer y dormir de la clase padre y cada uno tiene sus propios métodos adicionales. Por ejemplo, el perro puede vigilar la casa. Además, vemos como el loro es un tipo especial de pájaro que puede volar y hablar. Observa también cómo cada clase tiene su propio constructor, ya que los constructores nunca se heredan.

En la práctica, cuando hacemos programas, la herencia se traduce en que creamos objetos de una clase hija y podemos utilizar sobre ellos métodos de su clase padre como si fuesen métodos que hubiesen sido programados en la clase hija.

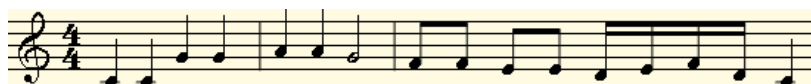
Por ejemplo, podemos hacer un programa que cree un Perro y lo pongamos a vigilar la casa 120 minutos y luego duerma 60 minutos, así:

```

3 public class Programa{
4     public static void main(String[] args){
5         Perro p = new Perro("Luna","Pequinés");
6         p.vigilarCasa(120); // método propio de la clase Perro
7         p.dormir(60);      // método heredado de la clase Animal
8     }
9 }

```

Ejercicio 31 : En la librería **Música** se encuentran clases para reproducir una sencilla partitura en un instrumento musical. Haz un programa que reproduzca en un piano la siguiente canción. Modifica el programa para que al finalizar, se vuelva a reproducir con una guitarra.



Recordatorio musical:

LAS NOTAS MUSICALES	DURACIÓN DE LAS NOTAS		
 DO RE MI FA SOL LA SI	REDONDA BLANCA	NEGRA CORCHEA SEMICORCHEA	

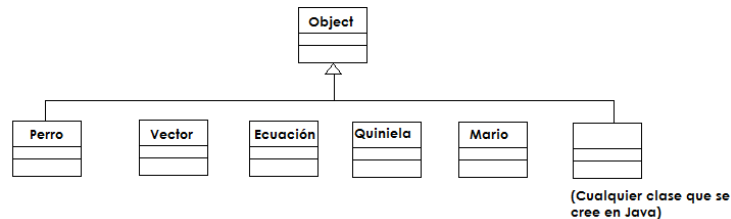
Ejercicio 32 : En la librería **ReproductorMúsica.jar** se encuentran clases para reproducir archivos mp3 y crear listas de reproducción que pueden guardarse en el disco. Su diagrama de clases y documentación vienen en el pdf que acompaña la librería. Realiza un programa que reproduzca en segundo plano (asíncrona) un archivo mp3 que tengas en tu disco duro. Una vez comenzada la reproducción, se mostrará el título de la canción, su autor y su duración.

Ejercicio 33 : Haz un programa que cree una lista de reproducción titulada “Lista de Ejemplo” y añádele tres canciones a ellas. El programa deberá guardar dicha lista en el disco duro (puedes inventar el nombre del archivo) y luego reproducir las tres canciones de forma síncrona.

Ejercicio 34 : Haz un programa que cargue la lista de reproducción guardada en el ejercicio anterior y la reproduzca de forma síncrona.

8.1.- La clase Object

En Java hay una clase especial denominada **Object** que es la clase padre de todas las clases de Java. Es decir, cualquier clase que exista en Java, da igual la persona o empresa que la haga, automáticamente heredar  de la clase Object:



La herencia de Object hace que todas las clases de Java hereden sus m todos. Los m todos que m s nos interesan de esta clase son:

- **boolean equals(Object o)**

Devuelve true si el objeto sobre el que se llama el m todo es **indistinguible** del objeto que se pasa como par metro. Por tanto, este m todo sirve para comparar si un objeto es un "hermano gemelo" de otro. Podr amos tener dos objetos diferentes, pero id nticos, y entonces el m todo equals devolver  true. La responsabilidad de que equals funcione correctamente depende de los programadores de clases.

Gracias a este m todo podemos comparar si dos String guardan el mismo dato. Por ejemplo, el siguiente programa devolver  true.

```
3 public class Programa{
4     public static void main(String[] args){
5         String a = "Hola";
6         String b = "Hola";
7         boolean iguales = a.equals(b);
8         System.out.println(" Son iguales? "+iguales);
9     }
10 }
```

- **int hashCode()**

Devuelve un n mero que identifica el objeto y a sus "hermanos gemelos". Esto significa que dos objetos que **equals** detecte como id nticos, deber n tener el mismo hashCode. La responsabilidad de que equals funcione correctamente depende de los programadores de clases. Por defecto, el m todo **hashCode** programado en la clase Object asigna un valor diferente a cada objeto.

- **String toString()**

Devuelve una representaci3n textual del objeto. Lo m s habitual es que sea un texto con el nombre completamente cualificado de la clase y su hashCode, pero en algunas clases es un texto que proporciona informaci3n sobre el objeto.

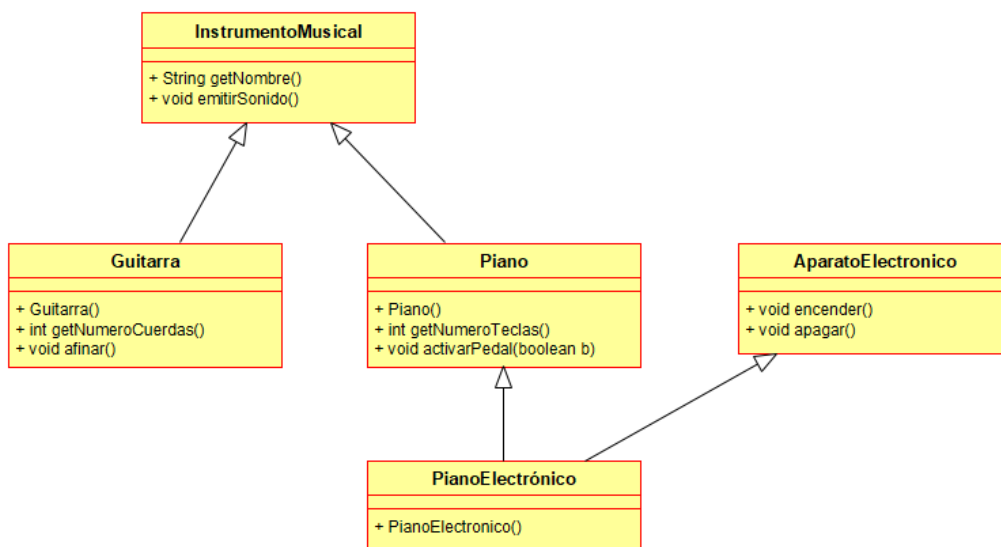
Ejercicio 35 : Realiza las siguientes tareas y responde las preguntas que se indican:

- a) Crea dos objetos diferentes ArchivoMP3, pero que usen la misma ruta de archivo.
 - a. Compáralos con el signo == como si fuesen tipos básicos. ¿Qué sucede?
 - b. Compáralos con el método equals. ¿Qué sucede?
 - c. Muestra los hashCode de dichos objetos. ¿Coinciden?
- b) Crea un objeto Reproductor y muestra por pantalla la salida de su método toString. ¿Qué aparece?
- c) Crea un objeto ArchivoMP3 y muestra por pantalla la salida de su método toString. ¿Qué aparece? ¿Tiene el mismo formato que la salida de toString de Reproductor? ¿a qué crees que puede ser debido?

9.- Interfaces

Como ya hemos visto, un problema de las clases es que solamente puede haber **herencia simple** entre ellas y eso no es del todo bueno, porque en la vida real hay situaciones donde un programador de clases puede tener casos en los que se dé herencia múltiple.

Por ejemplo, el siguiente diagrama muestra un ejemplo donde el programador de clases de la librería necesitaría hacer uso de herencia múltiple:



El diagrama anterior, tal y como está dibujado, no se podría programar en Java, porque las clases no admiten la herencia múltiple y en él vemos como la clase PianoElectrónico hereda a la vez de las clases Piano y AparatoElectrónico.

Para solucionar el problema y poder hacer diseños en los que si haya herencia múltiple, los creadores de Java han añadido a los tipos referencia una nueva categoría de tipos de datos muy parecidos a las clases, que se denominan **interfaces**.

O sea, los tipos referencia incluyen las clases y también las interfaces

9.1.- Las interfaces

Las interfaces son **tipos de datos referencia** que se parecen mucho a las clases, ya que también sirven para representar conceptos de la vida real:

Tipos básicos	Tipos referencia	
byte	String	■ clases
short	int[]	■ interfaces
int	Perro	
long	Balón	
double	Robot	
float	AparatoElectrónico	
boolean	Futbolista	
char	Limpiador	

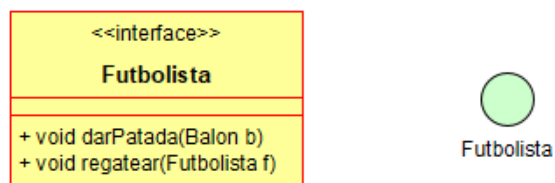
Al igual que les pasa a las clases, las interfaces también tienen métodos y las variables cuyo tipo de dato es una interfaz también son objetos.

En principio para un programador de aplicaciones no hay mucha diferencia⁸ entre clases e interfaces, salvo estas:

- **Las interfaces no tienen método constructor:** Por tanto, necesitaremos alguna forma diferente para obtener un objeto cuyo tipo sea una interfaz.
- **En la documentación se representan de forma diferente:** En los Javadoc las interfaces vienen indicadas con la palabra “interface”, o con su nombre en cursiva. Por ejemplo:



Cuando tenemos la documentación en papel, las interfaces admiten dos representaciones: como rectángulo con sus métodos y como un círculo con su nombre.

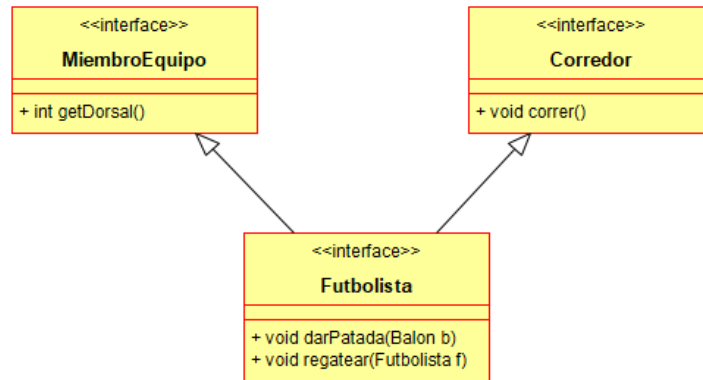


⁸ Para el programador de clases, las clases y las interfaces se programan de forma diferente, y además representan conceptos distintos de la vida real. Lo estudiaremos en el tema 5.

9.2.- La herencia múltiple

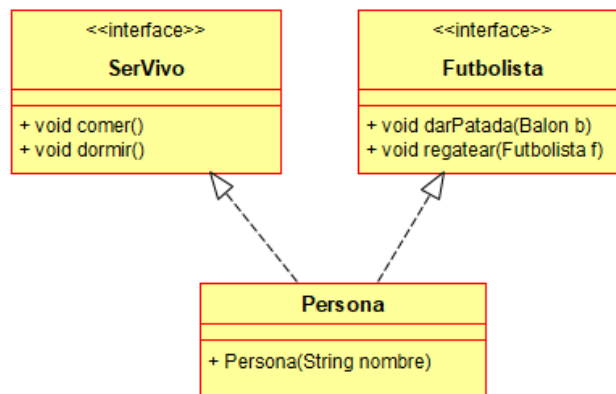
Como ya hemos comentado, las interfaces nacen para poder hacer herencia múltiple, que como recordamos, es una relación **es un** en la que hay varios padres.

- Podemos tener **herencia múltiple entre interfaces**, que la representaremos así:

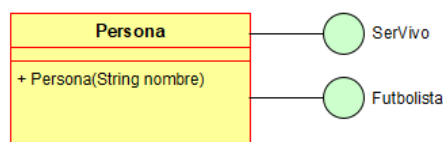


- También podemos tener herencia múltiple entre una clase y varias interfaces, pero a esta situación no se le llama herencia, sino que tiene otro nombre. Cuando una clase “hereda” de (una o varias) interfaces, diremos que “**la clase implementa las interfaces**”. Pese al cambio de nombre el concepto es el mismo, y es que hay una relación **es un** entre la clase y las interfaces, y por tanto, la clase recibe todos los métodos de las interfaces.

Para indicar la relación de implementación la flecha de la herencia se dibuja con líneas discontinuas:



Cuando la interfaz se representa como un círculo esta situación la podemos ver dibujada de otra forma, pegándole los círculos a la clase como si fueran “medallas”.



En los javadoc, podemos reconocer si una clase implementa interfaces mirando al principio, en la sección “**All implemented interfaces**”. También se pueden ver encontrando la palabra **implements** y viendo la lista de interfaces que aparece.

PACKAGE	CLASS	USE	TREE	DEPRECATED	INDEX	HELP
PREV CLASS	NEXT CLASS	FRAMES	NO FRAMES	ALL CLASSES		
SUMMARY: NESTED FIELD CONSTR METHOD DETAIL: FIELD CONSTR METHOD						

Class Persona

java.lang.Object
Persona

All Implemented Interfaces:
Futbolista, SerVivo

```
public class Persona
extends java.lang.Object
implements Futbolista, SerVivo
```

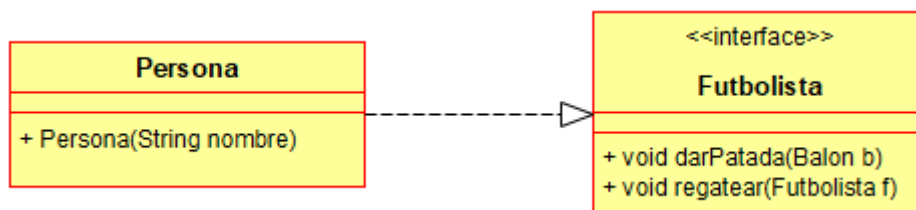
Esta clase representa una persona

← interfaces implementadas por Persona

9.3.- Las interfaces en los programas

Cuando hacemos un programa trabajaremos con las interfaces como si fuesen clases, pero con la diferencia de que no hay constructor. ¿Cómo se consigue entonces un objeto cuyo tipo sea una interfaz? En los siguientes ejemplos vamos a ver las dos situaciones habituales:

Ejemplo 1: Tenemos una clase que implementa una interfaz:



En este ejemplo no podemos crear directamente una variable de tipo Futbolista, pero como Persona es un Futbolista, entonces podemos crear una persona y guardarla dentro de una variable de tipo Futbolista, así:

```
1. Futbolista f = new Persona("Juan Diego");
```

Ahora, podemos seguir haciendo el programa y llamar a los métodos de Futbolista sobre la variable “f”, pero no a los métodos de Persona:

```

1. Balon b=new Balon();
2. Futbolista f = new Persona("Juan Diego");
3. f.correr(); // CORRECTO
4. f.darPatada(b); // CORRECTO
5. f.comer(); // INCORRECTO porque f es de tipo Futbolista y no SerVivo
  
```

A las personas que están aprendiendo a programar al principio les parece que las interfaces son peores que las clases, porque **perdemos métodos**. Como hemos visto en el ejemplo, la variable “f” es de tipo Futbolista y al guardar la Persona dentro de ella, perdemos el resto de los métodos de Persona. Por ejemplo, “f” ya no puede comer ni dormir.

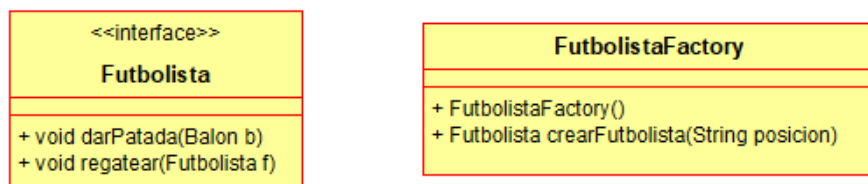
En realidad no existe este problema. Cuando queremos hacer un programa la mayoría de las veces no nos interesan todos los métodos de una clase, sino solo aquellos necesarios para resolver el problema⁹. Guardar los objetos en interfaces tiene dos ventajas:

- **Nos permite centrarnos solo en los métodos necesarios para hacer el programa.** Por ejemplo, si estamos haciendo un juego de futbol, no necesitamos que el futbolista pueda comer ni dormir, sino solo los métodos de Futbolista.
- **Mejora el mantenimiento ya que la clase es fácilmente reemplazable.** Por ejemplo, si el día de mañana aparece la clase Persona2, que es una versión optimizada de la clase Persona, solamente deberemos tocar una línea de código para actualizar el programa:

```
1. Balon b=new Balon();
2. Futbolista f = new Persona2("Juan Diego");
3. f.correr();
4. f.darPatada(b);
```

Por estos motivos, las interfaces se utilizarán siempre que sea posible. Durante el paso de los años y el desarrollo de proyectos, las empresas de programación han notado que el uso de interfaces mejora muchísimo el mantenimiento de los programas y hacen que un programa pueda crecer y alcanzar un tamaño muy grande. De hecho, hay empresas que solo usan interfaces en sus programas y la cantidad de clases es la mínima imprescindible¹⁰.

Ejemplo 2: Una clase tiene un método que devuelve objetos cuyo tipo de dato es una interfaz



En esta situación, si queremos obtener un objeto Futbolista nos bastará con obtener primero un objeto de la clase FutbolistaFactory y pedirle que nos cree un futbolista llamando a su método crearFutbolista de la forma habitual:

```
1. FutbolistaFactory ff = new FutbolistaFactory();
2. Futbolista a = ff.crearFutbolista("defensa");
3. Futbolista b = ff.crearFutbolista("delantero");
```

Una vez creados los objetos ya podemos utilizarlos en nuestro programa de la forma que hemos estudiado durante el tema, llamando a sus métodos.

⁹ Quedarnos solo con la cantidad de métodos necesarios y descartar el resto se denomina **abstracción**.

¹⁰ Las empresas que hacen esto siguen un principio llamado **program to interface**.

Ejercicio 36 : Usa la librería **Rol.jar** para hacer este juego:

- El jugador y el ordenador tienen 20 puntos de vida
- El jugador aplica una magia al ordenador y le resta los puntos de vida de la magia
- El ordenador (en caso de estar vivo) aplica una magia al jugador y le resta los puntos de vida de la magia
- El jugador (en caso de estar vivo) aplica una magia al ordenador y le resta los puntos de vida de la magia
- El ordenador (en caso de estar vivo) aplica una magia al jugador y le resta los puntos de vida de la magia
- Finalmente se muestra el ganador de la partida (el que más puntos de vida tenga)

En todo momento deberán mostrarse mensajes en pantalla con los puntos de vida de cada jugador y la magia que está siendo aplicada.

10.- Métodos estáticos

Hasta ahora hemos visto que los métodos son órdenes que damos a objetos concretos. Por ejemplo, si en un programa hay dos objetos Mario, puedo dar órdenes a cada uno de forma que el primero salte y el segundo se ponga a correr.

Estos métodos que hemos visto hasta ahora se llaman **métodos de instancia**, porque son métodos que aplicamos sobre objetos. Como es lógico, para poder usar un método de instancia, primero debemos tener un objeto sobre el que aplicarlo. No tiene sentido querer llamar a un método de instancia si no tenemos ningún objeto al que darle la orden.

Sin embargo, hay otro tipo de métodos llamados **métodos estáticos**. Los métodos estáticos son órdenes que le damos a la clase¹¹, como si la propia clase fuese un objeto.

Vamos a ver un ejemplo de método estático. Si miramos en la librería MarioLib, en la documentación de la clase Planta tenemos este método:

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type		Method and Description	
static int		<code>getTotalPlantas()</code> Devuelve el número total de plantas que hay en la pantalla	

Este método es estático porque aparece la palabra “static” a su izquierda. Lo que hace es decirnos cuántas plantas hay en la pantalla. Observa que este método es una pregunta que le hacemos a la clase, porque “¿cuántas plantas hay en la pantalla?” sólo nos la puede responder la clase. No es una pregunta dirigida a un objeto en particular.

¹¹ Aunque no lo decimos, las interfaces también pueden tener métodos estáticos.

Por tanto, para llamar a este método, utilizaremos el nombre de la clase (en este caso Planta) como si fuese una variable, así:

`int totalPlantas = Planta.getTotalPlantas();`
↑
↑
↑
↑
respuesta *clase* *método* *parámetros (no hay)*

El programa completo sería este:

```

1. public class Programa{
2.     public static void main(String[] args){
3.         Planta p1=new Planta(100,100);
4.         Planta p2=new Planta(150,250);
5.         Planta p3=new Planta(100,300);
6.         // ahora preguntamos a la clase Planta cuántas plantas hay
7.         int totalPlantas=Planta.getTotalPlantas();
8.         System.out.println("En la pantalla hay "+totalPlantas+" plantas");
9.     }
10. }
```

Ejercicio 37: En la “Librería Estándar de Java” está la clase **Math**, que sirve para realizar cálculos matemáticos. Consulta su documentación y calcula:

- a. La raíz cuadrada de 150
- b. 2 elevado a 9, y se divide entre el logaritmo neperiano de 6
- c. Coseno de 0.16 radianes.
- d. Seno de 45 grados (utiliza la clase **Math** para pasar de grados a radianes)

Ejercicio 38 : Haz un programa que calcule las dos soluciones de la ecuación $x^2 - 5x + 6$

Ejercicio 39 : Consulta el pdf de la “Librería Estándar de Java” y busca la clase **System**. Haz un programa que muestre por pantalla información sobre tu ordenador, de la forma que indica la siguiente imagen. (Nota: La carpeta de archivos temporales es la variable de entorno TMP).

```

Información sobre el ordenador:
- Sistema operativo: Windows 10
- Versión del sistema operativo: 10.0
- Arquitectura del sistema operativo: amd64
- Usuario actual: Juan Diego
- Carpeta de archivos temporales: C:\Users\JUAND~1\AppData\Local\Temp
```

Ejercicio 40 : Consulta el pdf de la “Librería Estándar de Java” y busca la clase **Integer**, que está en la hoja de “clases envoltorio”. Haz un programa que pregunte al usuario por teclado un número. El programa mostrará ese número escrito en binario y hexadecimal.

Ejercicio 41 : Consulta el pdf de la “Librería Estándar de Java” y busca la clase **Runtime**. Haz un programa que muestre por pantalla el número de procesadores de tu ordenador y la cantidad de memoria total asociada al proceso de Java por el sistema operativo.

Ejercicio 42 : Haz un programa que calcule el valor de $(0.38*4.93)^{-0.36}$ y lo muestre por pantalla. Redondea el resultado al entero más cercano y muéstralo también por pantalla.

Ejercicio 43 : Realiza un programa que genere un número aleatorio comprendido entre 29 y 65 y lo muestre por pantalla. *Nota: Hay dos formas posibles de hacerlo: con la clase **Math** y con la clase **Random** de la “Librería Estándar de Java”. Mira las dos y elige la que te resulte más fácil.*

Ejercicio 44 : Consulta el pdf de la “Librería Estándar de Java” y busca las clases **Math** y **NumberFormat**. Haz un programa que calcule el valor del número pi usando la siguiente fórmula: $\pi = 4 * \arctan(1)$ y muestre el valor usando 4 cifras decimales.

Ejercicio 45 : Consulta el pdf de la librería “java time” y busca las clases **LocalDate** y **DateTimeFormatter**. Haz un programa que muestre por pantalla la fecha actual en formato día/mes/año y después en formato día-mes-año.

Ejercicio 46 : Consulta el pdf de la librería “java time” y busca las clases **LocalDate** y **DayOfWeek**. Haz un programa que muestre por pantalla el nombre del día de la semana que será el 28 de febrero de 2100.

Ejercicio 47 : Consulta la “Librería estándar de Java” y busca las clases **Thread** y **Random**. Haz un programa que haga una pausa de una cantidad aleatoria de segundos entre 0 y 10. Consulta ahora el pdf de la librería “java time” y busca las clases **Instant** y **Duration**. Obtén un objeto Instant antes de que empiece la pausa y luego otro objeto Instant cuando la pausa termine. A partir de dichos dos objetos, obtén un objeto Duration y úsalo para mostrar cuántos segundos ha durado la pausa.

Ejercicio 48 : Consulta el pdf de la “Librería Estándar de Java” y busca la clase **Desktop**. Haz un programa que pregunte al usuario la ruta de un archivo. El programa abrirá dicho archivo usando para ello el programa que esté configurado en el sistema operativo.

11.- Generics

En este apartado vamos a ver un problema que aparece mucho en el mundo real y cómo se resuelve en lenguaje Java. Presentaremos el problema por medio de un ejemplo:

Supongamos que un programador de clases desea hacer una librería con una clase llamada **Bolsa**, que sirva para guardar objetos de un tipo concreto. Debido a que hay muchos tipos posibles de objetos, este programador debería hacer un montón de clases como estas:

- **BolsaString** → Clase que representa una bolsa para guardar String
- **BolsaMario** → Clase que representa una bolsa para guardar objetos de la clase Mario
- **BolsaPan** → Clase que representa una bolsa para guardar objetos de la clase Pan
- Etc

Aquí tenemos los diagramas de cómo podrían ser estas clases:

BolsaString	BolsaMario	BolsaPan
+ BolsaString() + void añadir(String s)	+ BolsaMario() + void añadir(Mario m)	+ BolsaPan() + void añadir(Pan p)

Como podemos imaginar, hacer una clase Bolsa por cada tipo de dato es una locura, ya que en la práctica ¡hay infinitos tipos de datos!

Ante este problema lo que nos gustaría sería poder hacer una sola clase Bolsa de manera que el tipo de dato de los objetos que se van a guardar dentro se pudiera elegir en el momento de crear un objeto Baolsa. En Java esto se consigue con el uso de **generics**¹².

Un **generic** es un tipo de dato “genérico” (normalmente llamado **T**) que aparece encerrado entre **< >** en el nombre de la clase, y que toma forma cuando creamos el objeto de la clase¹³.

O sea, la clase Bolsa que unifica a todas las bolsas posibles se va a llamar **Bolsa<T>**, y en el momento de crear una bolsa es cuando sustituimos T por el tipo de dato de los objetos que vamos a guardar en la bolsa.

Por ejemplo, la clase **Bolsa<T>** puede tener este aspecto:

Bolsa<T>
+ Bolsa() // crea una bolsa vacía + void añadir(T obj) // añade un objeto de tipo T a la bolsa

Como vemos, el generic T aparece encerrado entre <> formando parte del nombre de la clase y también aparece en los métodos indicando un tipo de dato genérico que tomará forma cuando hagamos el programa.

Si en un programa necesitamos crear una bolsa que guarde String, escribiremos:

```
1. Bolsa<String> b = new Bolsa<String>();
```

Observa cómo al escribir el tipo de dato de la variable concretamos el generic sustituyéndolo por el tipo de dato que nos interesa (en este caso T = String).

En realidad Java es lo suficientemente inteligente¹⁴ como para que no haga falta poner el tipo de dato en la parte del “new”. O sea, podemos crear la bolsa así:

```
1. Bolsa<String> b = new Bolsa<>();
```

¹² Los **generics** aparecieron en el lenguaje Java en el año 2004 (versión Java 5)

¹³ Aunque no lo decimos, las interfaces también pueden tener generics.

¹⁴ El símbolo <> sirve para inferir el generic y se denomina **operador diamond**. Apareció en Java 7 (2011)

Una vez creada la bolsa de String, podemos añadir a esa bolsa todos los objetos String que queramos, por ejemplo:

```
1. Bolsa<String> b = new Bolsa<>();
2. b.añadir("Hola");
3. b.añadir("Adios");
```

Si en lugar de String necesitamos una bolsa que guarde muñecos de Mario, haremos:

```
1. Bolsa<Mario> m = new Bolsa<>();
```

Podemos añadir a esa bolsa todos los muñecos de Mario que queramos, bien directamente, o bien muñecos que estén previamente guardados en variables

```
1. // creo una bolsa para guardar muñecos de Mario
2. Bolsa<Mario> m = new Bolsa<>();
3. // añado un muñeco de Mario que está en (120,90) directamente
4. m.añadir(new Mario(120,90));
5. // creo un muñeco de Mario en una variable
6. Mario a = new Mario(300,200);
7. // añado el muñeco de la variable a la bolsa
8. m.añadir(a);
```

11.1.- Los tipos envoltorio

Los generics tienen una restricción muy importante:

Los generics solo pueden sustituirse por tipos referencia, o sea, NO se admiten tipos básicos

Por tanto, si en ejemplo de antes quisiéramos hacer una bolsa que guardara números enteros, tendríamos un problema, ya que la siguiente línea nos daría error de compilación:

```
public static void main(String[] args) {
    Bolsa<int> b=new Bolsa<>();
}
```

unexpected type
required: reference
found: int

(Alt-Enter shows hints)

Como no es posible usar tipos básicos en un generic, los creadores de Java introdujeron unas clases llamadas **tipos envoltorio (o wrappers)**.

La idea es que cada tipo básico va a tener una clase (llamada su “tipo envoltorio”) que lo representa. Aquí tenemos la tabla con los tipos envoltorio de cada tipo básico:

Tipo básico	Tipo envoltorio
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Lo bueno de los tipos envoltorios es que Java es capaz de convertir automáticamente¹⁵ entre el tipo básico y su envoltorio. Por ejemplo, para crear un Integer no es necesario usar ningún constructor, sino que podemos asociarle directamente un número de su tipo básico así:

```
1. Integer a = 5;
```

De hecho, ya que Java convierte automáticamente al tipo correcto los tipos básicos y sus envoltorios, podemos hacer operaciones donde se mezclan tipos básicos y envoltorio, así:

```
1. Integer a = 5;
2. int b = 10;
3. Integer c = 4;
4. int d = 8;
5. Integer x = ((a + b) * c) - d; // también podríamos hacer int x = ((a+b)*c)-d;
```

Si volvemos a la clase Bolsa<T> de los ejemplos anteriores, aplicando los tipos envoltorios ahora sí que podemos crear una bolsa de cualquier tipo básico que queramos. En realidad, no estamos creando ninguna bolsa asociada a un tipo básico, sino a su tipo envoltorio.

Por ejemplo, si necesitamos crear una bolsa para guardar números enteros, haremos:

```
1. Bolsa<Integer> b = new Bolsa<>();
```

Y ahora podríamos llenar la bolsa con cualquier cantidad de números enteros, así:

```
1. Bolsa<Integer> b = new Bolsa<>();
2. b.añadir(4);
3. b.añadir(6);
4. b.añadir(2);
```

Podemos hacer lo mismo con cualquier otro tipo básico. Por ejemplo:

```
1. Bolsa<Boolean> b = new Bolsa<>();
2. b.añadir(true);
```

☞ **¿Por qué no usamos siempre tipos envoltorio y nos olvidamos de los tipos básicos?** Porque los tipos envoltorio son objetos y por tanto, ocupan muchísima más memoria que los tipos básicos. Además, al trabajar con los tipos envoltorios (llamar a sus métodos, hacer cambio automático entre un tipo básico y referencia, etc) se consume tiempo de procesador.

Ejercicio 49 : Usa las clases de la librería **DoctorLib.jar** para hacer un programa que haga esto:

- a) Crea un perro, un gato y una paloma (inventa todos los nombres)
- b) Haz que el perro ataque a la paloma y al gato
- c) Crea un doctor que sepa curar gatos y haz que cure al gato, comprobando que se ha curado
- d) ¿Puedes hacer que ese doctor pueda curar a la paloma?
- e) Crea un doctor que sepa curar palomas y cure a la paloma.

¹⁵ Esta característica se denomina **autoboxing** y apareció en Java 5 (2004). Antes de su aparición, para trabajar con tipos envoltorios era necesario crearlos con su constructor y llamar a sus métodos. Gracias al autoboxing los tipos envoltorios se manejan como si fuesen tipos básicos, simplificándose su uso.

Ejercicio 50 : En el pdf “programación funcional.pdf” está la clase **Optional<T>**. Consulta su documentación para hacer este programa:

- Se creará una variable de tipo `Optional<Double>` llamada **resultado** y se le dará de inicio el valor `null`.
- Se pedirán dos números enteros por teclado llamados `dividendo` y `divisor`.
- Si el divisor es cero, en la variable **resultado** se guardará un optional vacío.
- Si el divisor es distinto de cero, se calculará la división y su resultado se guardará en la variable **resultado** dentro de un `Optional` que lo envuelva.
- Por último, el programa comprobará si el `Optional` de la variable **resultado** está vacío, y en caso contrario, extraerá de él el resultado de la división y lo mostrará por pantalla.

12.- Constantes

A lo largo del tema hemos visto que una clase tiene métodos de todo tipo: constructores, métodos de instancia y métodos estáticos.

Sin embargo, además de métodos, las clases¹⁶ también pueden tener **constantes**. Las constantes son valores u objetos que están creados dentro de la clase y que tienen un nombre especial con el que reconocerlo. Usar este nombre nos ayuda a utilizar el valor definido en la constante de manera directa en un programa.

El ejemplo típico de constante es el número pi. Como sabemos, dicho número es:

$$\pi = 3,14159265358979323846 \dots$$

Si tenemos que hacer un programa donde aparezca este número, tenemos dos opciones:

- Crearlo y guardarlo en una variable o constante de nuestro programa: Esto está bien, pero nos obliga a tener que aprendernos su valor y a tener que repetirlo en todos los programas que hagamos
- Utilizamos la constante **Math.PI**, que guarda un valor estándar del número pi.

Como nos dice el último punto, la clase `Math` que hemos visto en los ejercicios anteriores, incluye una copia del número pi, llamada `Math.PI`. En cualquier momento de nuestro programa podemos utilizarla. Por ejemplo, así:

```
1. public class Programa{
2.     public static void main(String[] args){
3.         System.out.println("El número pi es: "+Math.PI);
4.     }
5. }
```

¹⁶ Aunque no lo decimos, las interfaces también pueden tener constantes.

Como es lógico, para conocer las constantes que puede tener una clase, deberemos irnos a su documentación, y buscarlas en la sección “**field summary**”. Deben aparecer marcadas como “**static final**”. Por ejemplo, en el javadoc de la clase `Math` se puede ver:

Field Summary

Fields	
Modifier and Type	Field and Description
static double	E The double value that is closer than any other to <i>e</i> , the base of the natural logarithms.
static double	PI The double value that is closer than any other to <i>pi</i> , the ratio of the circumference of a circle to its diameter.

Aquí vemos como las constantes `Math.PI` y `Math.E` son de tipo `double`, pero en realidad pueden tener cualquier tipo de dato, o incluso ser objetos. Por ejemplo, en el javadoc de la clase `Color` de la librería “`java 2d`” podemos ver constantes que representan los colores y son objetos de la clase `Color`.

Field Summary

Fields	
Modifier and Type	Field and Description
static Color	black The color black.
static Color	BLACK The color black.
static Color	blue The color blue.
static Color	BLUE The color blue.

Las constantes también se usan para identificar de manera sencilla los valores que podemos pasar a los métodos y así evitar errores. Los siguientes ejercicios son un ejemplo de esa situación.

Ejercicio 51 : En la librería “Objetos Sencillos” está la clase **MarcadorMejorado**. Haz un programa en el que se cree un partido de baloncesto para el partido que van a jugar el Estudiantes y el CB Granada. Registra las siguientes canastas y muestra el nombre de los equipos, los puntos que tiene cada uno, el nombre del equipo que va ganando y el que va perdiendo. Compara el código con el del ejercicio 12. ¿Cuál crees que es mejor?

E ->2, CB -> 3, E->2, CB->2, CB->3, E->1, E->1, CB->2

Ejercicio 52 : En la librería “Efecto Imagen” se encuentra la clase **EfectoBuilder**, que sirve para aplicar efectos a una imagen. Consulta su documentación y realiza un programa que haga esto:

- Pregunte al usuario la ruta de un archivo con una imagen
- Utiliza la clase **ImageIO** de la librería “`java 2d`” para cargar la imagen
- Utiliza la clase **EfectoImagen** para aplicar, en este orden, los efectos “invertir”, “blurred” y “escala de grises” a la fotografía.
- Cuando hayas terminado de aplicar los efectos, muestra una ventana en la que se vea la imagen final.

13.- Tipos de dato referencia

Como ya sabemos, los tipos de datos se dividen en dos categorías: básicos y referencia. A lo largo del tema han ido saliendo algunas diferencias entre ellos, como por ejemplo:

- Los tipos básicos son siempre los mismos, mientras que los tipos referencia se pueden ampliar mediante librerías.
- Los tipos básicos admiten operaciones, mientras que los tipos referencia tienen métodos.
- Las variables de tipo referencia pueden dejarse vacías, asignándoles el valor null. Las variables de tipo básico siempre deben tener un valor guardado en ellas.

Sin embargo, en este apartado vamos a estudiar la principal diferencia entre los tipos básicos y los tipos referencia, que está en cómo se guardan internamente en la memoria ram del ordenador.

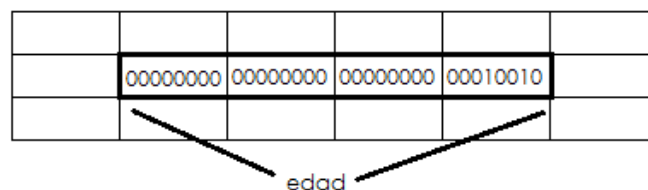
13.1.- Representación en memoria de los tipos básicos

Cuando en un programa creamos una variable de un tipo básico, el ordenador reserva en la memoria tantos bytes como necesita su tipo (por ejemplo, el int necesita 4 bytes) y rellena en dichos bytes con el valor binario que le damos a la variable.

Por ejemplo, si en un programa se escribe:

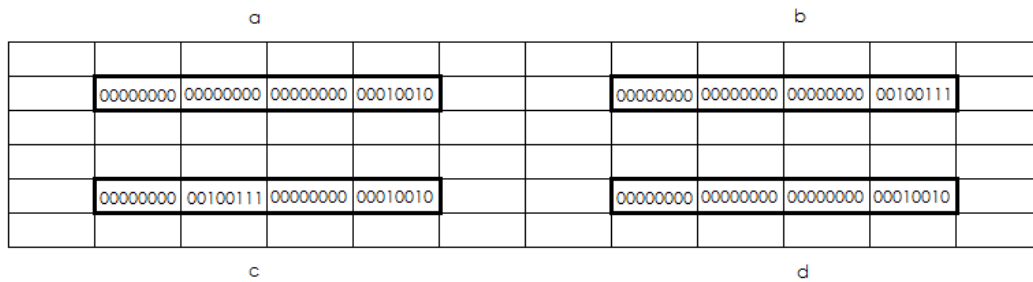
```
int edad=18;
```

En la memoria del ordenador se reserva una zona de 4 bytes en la que se almacena el número 18 escrito en binario:



Si tenemos dos variables de tipo básico y las comparamos con ==, el ordenador observa si el contenido de las zonas de memoria coincide byte a byte y así es capaz de saber si son iguales o diferentes.

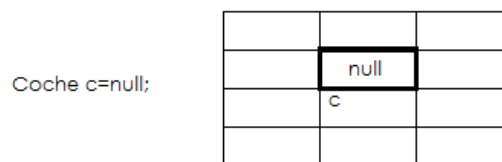
En la siguiente imagen, las variables "a" y "d" son iguales porque su contenido coincide byte a byte. Por tanto, **a==d** sería true, mientras que las demás comparaciones, como por ejemplo **a==b** o **a==c** serían false.



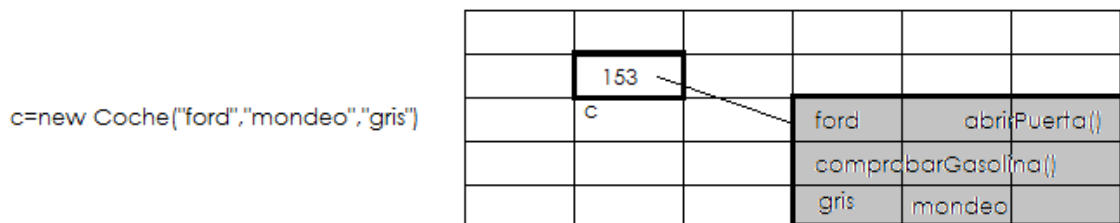
13.2.- Representación en memoria de los tipos referencia

La representación en memoria de los objetos es muy diferente de la de los tipos básicos. Es necesario conocerla, ya que en la práctica pueden presentarse situaciones extrañas que solo tienen explicación si conocemos lo que ocurre internamente.

Cuando se define por primera vez una variable de un tipo referencia, se reserva una posición de memoria y se guarda en ella el valor **null**, ya que el objeto estará sin asignar.



Cuando se utiliza el constructor para dar vida a un objeto, se crea un bloque en la memoria donde se almacenan las características del objeto en sí, y la variable de partida se rellena con **la dirección de memoria**¹⁷ donde se ubica dicho bloque.



Esta forma de actuar, que obedece a motivos de eficiencia, puede producir situaciones extrañas a la hora de comparar objetos, como lo que ocurre en el siguiente ejemplo:

```

1. public class Programa{
2.     public static void main(String[] args){
3.         Coche a = new Coche("ford", "mondeo", "gris");
4.         Coche b = new Coche("audi", "a4", "negro");
5.         Coche c = new Coche("ford", "mondeo", "gris");
6.         Coche d = new Coche("audi", "a4", "negro");
7.         System.out.println(a==c);
8.         System.out.println(b==d);
9.         a=b;
10.        System.out.println(a==b);
11.    }
12. }

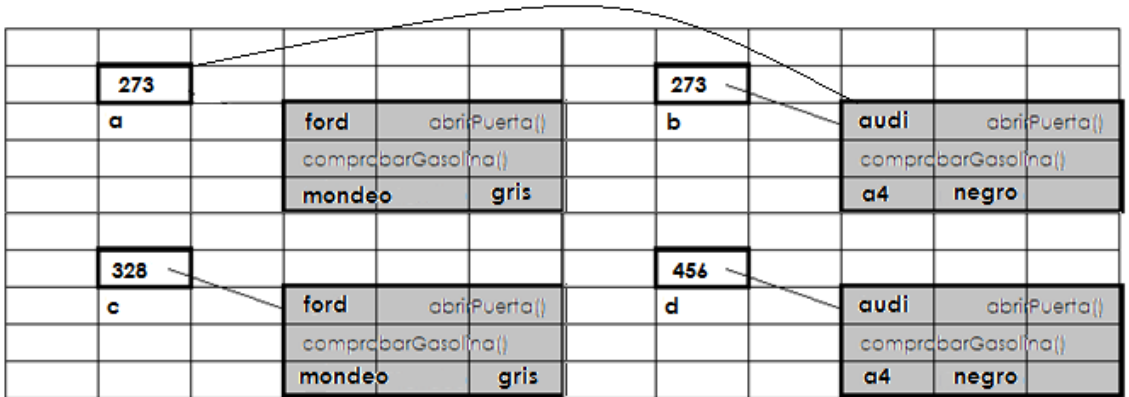
```

¹⁷ En los lenguajes C y C++ las variables que almacenan una posición de memoria se denominan **punteros**.

150	a	ford	abrirPuerta()	comprobarGasolina()	mondeo	gris
273	b	audi	abrirPuerta()	comprobarGasolina()	a4	negro
328	c	ford	abrirPuerta()	comprobarGasolina()	mondeo	gris
456	d	audi	abrirPuerta()	comprobarGasolina()	a4	negro

Por tanto, la comparación `a==c` mostrará `false`, aunque nosotros veamos que los dos coches sean iguales. Por este motivo, no se puede usar `==` para comparar tipos referencia, porque lo que estaríamos comparando serían sus direcciones de memoria. Para comparar si dos objetos son “indistinguibles”, debemos utilizar el método `equals`¹⁸, que se hereda de la clase `Object`. Por ejemplo, `a.equals(c)` devolvería `true` si la clase `Coche` estuviese bien hecha.

Si seguimos analizando el programa llegamos ahora a la línea 12. Allí asignamos el valor de la variable “b” a la variable “a”. Haciendo esto, el valor antiguo de la variable “a” (el 150) se pierde y se guarda en ella el valor que tiene la variable “b” (el 273). La memoria queda así:



48

En esta situación observamos que hay dos variables llamadas “a” y “b” que apuntan al mismo bloque de memoria. Si pasamos a la línea 13 y comparamos **a==b**, entonces estamos comparando 273 con 273, por lo que el resultado es true.

La conclusión final que lo resume todo es que:

- **a==b** compara las direcciones de la memoria del objeto apuntado por a y b. Por tanto, solo dará true cuando ambas variables apunten al mismo objeto de la memoria. Su valor en objetos distintos, aunque sean idénticos, será false.
- **a.equals(b)** compara si dos objetos son idénticos. Por tanto, será true cuando tengamos objetos diferentes, pero indistinguibles uno de otro. No obstante, su correcto funcionamiento depende del programador de la clase que lo utiliza.

Si observamos la imagen anterior, podemos ver que el bloque de memoria del antiguo objeto guardado en “a” se ha quedado suelto, sin ninguna variable que lo apunte. En el lenguaje Java existe un proceso en segundo plano denominado **Recolector de Basura** (Garbage Collector) que entra en funcionamiento de forma automática cada pocos milisegundos y borra de la memoria los bloques libres que han dejado de ser apuntados. Esto facilita el trabajo al programador, que no debe preocuparse por liberar los bloques libres¹⁹, pero hace que la ejecución de los programas sea un poco más lenta, por tener activo un proceso en segundo plano además del programa.

¹⁹ En lenguajes como C o C++ no existe la recolección de basura y el programador debe liberar manualmente los bloques libres. En caso de no hacerlo, estos bloques no podrán ser reutilizados y disminuirá la cantidad de memoria disponible en el ordenador.