

1. INTRODUCCIÓN A LA PROGRAMACIÓN

Programación es la escritura de comandos (también podemos llamarlas instrucciones u órdenes) que el ordenador, al interpretarlos, los convertirá en un programa. Mediante el uso de un **compilador**, la máquina será capaz de entender las secuencias de estos comandos.

La programación no es nueva, podemos irnos a 1800 para ver las primeras programaciones. Destacar a **Ada Lovelace**, que fue la **primera persona en crear un algoritmo que procesaría una máquina**.

En la **década de los '50** encontramos los **primeros lenguajes de uso común**, que se siguen utilizando hoy día. **Los lenguajes de hoy vienen de ellos**.

A finales de esta década destacamos el **lenguaje Cobol (muy utilizado en banca y seguros)** y que hoy en día se sigue utilizando con evoluciones.

En la **década de los '70** hubo una gran evolución. Podemos resaltar el **lenguaje B, precursor del lenguaje C**, tan utilizado hoy. También en esta década se creó el **lenguaje Pascal (se sigue utilizando)** y el **lenguaje SQL**, el cual está hoy en día en la mayoría de aplicaciones web que utilizamos.

En la **década de los '80** apareció el **lenguaje C++ (lenguaje utilizado para software de gran rendimiento como Google Chrome)**. También es destacable el **lenguaje ADA (utilizado para software militar y aeroespacial)**.

En la **década de los '90**, con la extensión a todos de Internet, **aparecieron los lenguajes** que todos conocemos, **como Python muy utilizado en IA**. Destacar **HTML** (lenguaje de marcas para web), **Java**, **JavaScript** o **PHP**.

Todos los lenguajes los podemos clasificar en dos tipos:

- **Lenguajes compilados**. Mediante una herramienta llamada compilador, se transforman desde un lenguaje fuente (secuencia de comandos) a la representación binaria que es comprendida por el procesador. Es decir, el compilador coge nuestras instrucciones, lo convierte en un formato que entiende la máquina y lo ejecuta tal cual.
- **Lenguajes interpretados**. Utilizan otro programa para ejecutarlos directamente. Es decir, no se transforman directamente en un código que entiende la máquina. JavaScript, Java, Python son ejemplos de este tipo de lenguaje.

Podemos diferenciar una gran cantidad de aplicaciones diferentes como, apps móviles, navegadores web, sistemas operativos, software de un coche, etc. Sin embargo, **podemos clasificarlas en tres tipos**:

- **Aplicaciones web.** Funcionan mediante acciones, es decir, son interactivas. Según la acción que proponemos, pasa una cosa u otra.
- **Aplicaciones de escritorio.** Se orientan a eventos, es decir, cuando enviamos una acción, la aplicación responde a esa acción (evento).
- **Aplicaciones móviles.** Se orientan a vistas, es decir, en este tipo solo tenemos una pantalla general en la que podemos hacer una única acción.

Estructura de una web

Las aplicaciones web se estructuran siguiendo cierto patrón. Por una lado, tenemos **la parte que vemos de la aplicación web (front-end)** y, por otro lado, tenemos **la parte que no vemos de la aplicación web, lo que se ejecuta en los servidores (back-end)**.

El **front end** se divide en tres capas:

- **Capa HTML.** Estructura de la página web.
- **Capa CSS.** Da estilo a la página, es decir, el formato a la página (tipografía, colores, etc.).
- **Capa de funcionalidad.** Se implementa con **JavaScript**. La parte que **comunica las órdenes** entre el front-end con el back-end, por ejemplo, guardar un archivo.

Estructura de una aplicación de escritorio

Podemos realizar diversos eventos desde la misma aplicación. Entendemos **eventos** cuando hacemos clic en una parte de la aplicación y ésta responde. Por ejemplo, si clicamos en minimizar se genera el evento minimizar.

Este tipo de aplicaciones siempre están "escuchando" con el objetivo de reaccionar cuando creamos un evento.

Estructura de una aplicación móvil

Se orientan a **vistas**, es decir, únicamente tenemos una vista de la aplicación (pantalla) en la que podemos ver sólo esa pantalla. Si cambiamos de vista (pantalla) podremos realizar otras acciones, pero siempre cambiando de vista, **nunca podremos tener varias vistas a la vez**.

Indistintamente del tipo de aplicación (web, escritorio o móvil), **las aplicaciones pueden ser de cliente o de cliente/servidor**. Son las aplicaciones que se comunican con un tercero.

Un ejemplo de aplicación cliente/servidor es Twitter. Yo como usuario, envío un tweet mediante la aplicación cliente al servidor. Éste tweet llega a mis followers a través de la aplicación del servidor.

Por el contrario, la agenda (siempre que no esté sincronizada en la nube) sería una aplicación cliente únicamente, ya que solo podríamos ver esa agenda desde un único dispositivo.

¿Qué es una API?

Application Programming Interfaces o Interfaz de Programación de Aplicaciones son comandos que enviamos al servidor y que éste va a interpretar y entregar una respuesta, es decir, las peticiones que enviamos al servidor. Por ejemplo, utilizar el buscador para encontrar una persona con cierto nombre y apellidos en Instagram.

En definitiva, una API es lo que se utiliza para mantener la comunicación coherente entre el cliente y el servidor.

Podemos encontrar dos tipos de APIs:

- **API Pública.** Todos los clientes tienen acceso a ella.
- **API Privada.** Las que conocen los desarrolladores y unos pocos clientes.

¿Qué es una librería externa?

Conjunto de funciones para un lenguaje de programación. Estas librerías aportan funcionalidad, es decir, ayudan a mejorar la comunicación con la API, ya que da hecho mucha parte del desarrollo y simplifica el trabajo.

¿Qué es una variable?

Es el nombre que le damos a la dirección de la memoria. Como su nombre indica, puede variar, por tanto, el nombre puede cambiar dependiendo lo que tenemos guardado, así como los datos que hay guardados.

Las variables que no pueden variar se denominan **constantes**. Una vez hemos adjudicado un dato a la variable, éste, durante la ejecución del programa, no puede cambiar.

¿Cómo funciona la memoria?

Entendemos la memoria como el almacén donde se guardan todos los datos necesarios para ejecutar un programa. La memoria es finita, cuando se llena ningún programa podrá ejecutarse, ya que no podrá guardar más datos.

Conocemos como direcciones de memoria a todas las posiciones que tiene una memoria, es decir, cada lugar donde se guardan los datos. Para encontrar los datos deseados, debemos nombrar correctamente cada dirección de memoria.

2. PROGRAMACIÓN. VARIABLES Y CONSTANTES

Tipos de datos

- **Tipos de datos primitivos.** Son los tipos más básicos que nos aporta cualquier lenguaje de programación.
- **Tipos compuestos (o complejos).** Se nutren de los datos primitivos, los necesitan.

¿Para qué están los tipos de datos?

Hay dos paradigmas adicionales en el mundo del desarrollo:

- **Lenguaje tipado.** Exige que declaremos el tipo de dato que vamos a almacenar en memoria.
- **Lenguaje no tipado.** Determina sobre la marcha (ya sea mediante compilación o interpretación) que tipo de dato vamos a almacenar en memoria de forma implícita.

Destacar, que **hay lenguajes que son fuertemente tipados** y que a su vez cuando utilizamos variables no hace falta que pongamos el tipo de dato.

¿Qué es un tipo de dato?

Es una forma de decirle a nuestro programa cómo va a ser el valor que vamos a guardar en memoria, ya que podemos guardar valores de diferente tipo. Recordemos que **hacemos referencia a la memoria mediante variables** (no lo hacemos con direcciones).

En esta zona de memoria donde guardamos la variable pueden ser números enteros, que serían los **tipos numéricos enteros** y dentro de este **hay cuatro**:

- **byte.** Este tipo de dato ocupa **1 byte** (8 bits, 2^8 valores), por lo que podríamos representar desde el valor 0 al 255 ó -128 al 127. **En caso de utilizar un valor por debajo o por encima de los valores permitido provocaremos lo que se denomina un overflow** y depende del lenguaje o del compilador lo que puede ocurrir. Un overflow puede provocar algún comportamiento inesperado o que empiece a contar desde el principio.
- **short (o integer corto).** Ocupa **2 bytes** (16 bits, 2^{16} valores) y hace posible representar cualquier valor en el rango -32.768 al 32.767 o del 0 al 65.535.

En Java no hay una forma clara de decir que los tipos son con signo (**signed**) o sin signo (**unsigned**), por esto en ocasiones pueden ir de “-X” al “Y” o de “0” al “X”. En otros lenguajes es más sencillo, con un modificador le decimos que queremos contar del “0” al “X”, pero en otros lenguajes no lo podemos

hacer tan fácil y por defecto es del “-X” al “Y”. Java por ejemplo, es un lenguaje de los que no podemos decirle si es con signo o sin signo.

- **int (o integer)**. Ocupa **4 bytes (32 bits)**, 2^{32} valores) de almacenamiento y es el tipo de dato entero más empleado. El rango de valores que puede representar va de -2^{31} a $2^{31}-1$.
- **long**. Es el tipo entero de mayor tamaño, **ocupa 8 bytes (64 bits)**, 2^{64} valores), con un rango de valores desde -2^{63} a $2^{63}-1$.

Tenemos que **elegir siempre el tipo de dato que más se acomode a lo que vamos a necesitar, para hacer un uso óptimo de la memoria**. Cuando no sabemos que valor vamos a tener utilizamos un **int**.

Para representar números decimales usamos los **tipos numéricos en punto flotante**:

- **float**. Conocido como tipo de precisión simple, **emplea un total de 4 bytes (32 bits)**, 2^{32} valores).
- **double**. Sigue un esquema de almacenamiento similar al anterior, pero **usando 8 bytes (64 bits)**, 2^{64} valores).

Otros tipos de datos son:

- **char**. Los caracteres también son un tipo de dato, un único carácter es un tipo de dato, el tipo de dato de un único carácter. Estos caracteres pueden ser tan simples como la letra “a” (que equivale al 65 en decimal) o puede ser el carácter “NUL” (que equivale a 0 en decimal). Cuando en un editor de texto pulsamos intro se dan dos caracteres que son el LF (line feed que equivale al 10 en decimal) que es un salto a la línea siguiente y el CR (carriage return que equivale al 13 en decimal) que es un retorno de carro.
- **string (o cadenas de texto)**. Es un tipo de dato que **representa una serie de caracteres**. En un **string**, cada carácter de la cadena también es un tipo de dato **char**.

Podemos ver una lista de caracteres [ASCII](#) estándar con los 256 primeros caracteres. Además, tenemos los [Unicode](#) en el que se incluyen los emojis, también son un carácter (no son una cadena de texto).

Los **booleanos son otro tipo de dato**. El tipo es **boolean** **no está soportado en todos los lenguajes de programación** y son un tipo de dato muy especial, que puede contener dos valores, **true (verdadero)** o **false (falso)**.

Las expresiones **boolean** están en cualquier lenguaje de programación aunque no se soporte el tipo de dato, están en todas partes. En las condicionales en Java, todas las evaluaciones tienen que devolver dos cosas, verdadero o falso.

Al declarar variables tenemos que **empezar a nombrarlas por una letra o guión bajo, nunca por un número.**

Los **tipos de datos complejos** se usan mucho y tenemos los siguientes:

- **array (también arreglos o matrices).** Es una **colección de elementos con un mismo tipo de dato almacenado en posiciones de memoria consecutivas.** Se declara:

```
// Declaramos un array con tipos de datos int
int miArray[];
miArray = new int[10];
miArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Declaramos un array con tipo de datos char
char miChar[];
miChar = new char[18];
miChar = { 'h', 'o', 'l', 'a', 'm', 'u', 'n', 'd', 'o' };

// Mostramos el resultado en pantalla
System.out.println(miChar);
```

En el caso de querer incluir un 10 en el array (con tipos de datos **char**) tendría que usar un carácter especial como el `\n` que equivale a 10 y solo utiliza un carácter, ya que 10 utiliza dos caracteres y esto sería una cadena. El carácter Unicode `\u2764` es un corazón y ocupa un carácter.

De todos los tipos de datos primitivos se puede hacer un **array**.

Objetos

Un objeto es una **entidad que representa algo en el mundo real.** Los objetos **pueden tener propiedades y métodos** (que en otros lenguajes se pueden llamar funciones, en Go se llaman funciones en estructuras).

Un objeto **hace referencia a una posición en la memoria donde hay cierta información** almacenada (que puede ser consecutiva o no).

Por ejemplo, el objeto cuadro tiene unas propiedades (un ancho, un alto, una serie de colores,...) y no tiene métodos porque aparte de verlo no podemos hacer nada más.

También tenemos el objeto coche, tiene unas propiedades (tiene un color, unas puertas, un tipo de carburante, puede tener cuatro ruedas,...) y también tiene una serie de métodos (encender motor, apagar motor,...).

En el objeto Sr. Potato tenemos una serie de propiedades (tiene gorrito, unas orejas, una nariz, una boca,...) y métodos (quitar nariz, poner nariz, quitar ojos, poner ojos,...).

Hay otros tipos de datos que son más potentes pero entramos en el mundo de las **estructuras de datos**. Son **más complejos de utilizar** y requiere que tengamos cierta idea cómo funciona la memoria del ordenador.

Las **tuplas** son estructuras de datos inmutables, es decir, que no se pueden modificar después de su creación, pero permite extraer porciones y permiten comprobar si un elemento está en la tupla. También pueden utilizarse como clave en un diccionario de datos.

Los **diccionarios** tienen una **clave asociada a un valor** (una palabra y la definición de esa palabra). Muchos lenguajes las tienen inherentes y otros no.

Estas **estructuras de datos** (datos complejos) **están compuestas por tipos de datos básicos (o tipos primitivos)**.

Los **sets** son **colecciones de elementos**, y tenemos **dos tipos** de datos:

- **lista** es una **sucesión de elementos (del mismo tipo de dato) que pueden** estar repetidos.
- **colección** es una **sucesión de elementos (del mismo tipo de dato) que no pueden** estar repetidos.

Sobre la práctica son exactamente lo mismo, pero varían cuando se ejecutan.

Ejemplo de una lista (aunque se pueden crear de muchos otros tipos):

```
// Creando una lista
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        List<String> nombres = new ArrayList<>();
    }
}
```

Hay muchos más tipos como pueden ser los **mapas hash** muy útiles por su forma de acceso y la velocidad de acceso.

3. FUNCIONES

Una **función** nos evita tener que repetir código. Dentro de la función puede ir cualquier sintaxis del lenguaje.

Cuando en un programa tenemos que hacer muchas veces la misma tarea (escribir el mismo código), entonces creamos una función.

La ventaja de tener funciones es que al solucionar un fallo en el código de dicha función, lo tendremos solucionado en cada punto donde se haya usado la función. Esto evita tener que modificar mucho código.

Cada lenguaje tiene una forma de definir las funciones.

Las funciones tienen una cosa que se llama firma, prototipo o signature. Este prototipo consiste en:

- Como se llama la función
- Qué valores va a aceptar la función
- Qué tipo de datos va a retornar la función (en caso de que retorne alguno)

Ejemplo

```
public class Main {  
  
    public static void main(String[] args) {  
        int resultado = 0;  
        resultado = suma (4, 2);  
        System.out.println(resultado);  
    }  
    public static int suma(int a, int b) {  
        return a + b;  
    }  
}
```

La función con nombre **suma** va a devolver un tipo de dato **int** (integer o entero). Cuando invoque a la función le tengo que pasar los parámetros (**int a, int b**) del mismo tipo que vienen en la función, además la función internamente asume que son variables y retorna la suma de a+b (**return a + b**).

También **podríamos aceptar de un tipo de dato en los parámetros y devolver otro tipo de dato distinto**.

Ejemplo

```
public static String suma(int a, int b) {  
    return a + b;  
}
```

Cuando queremos que una función no devuelva nada, usamos el tipo de dato `void` que significa vacío o nulo.

Usamos una función que devuelva algo cuando en otra parte de mi programa espero invocar a esa función para que genere un resultado que voy a utilizar a posteriori. Y en el caso de una función que no devuelva nada, es porque invocamos a la función para que haga lo que tenga que hacer y no esperamos ningún resultado.

La función `main` es especial en muchos lenguajes, porque cuando se ejecuta el programa internamente lo primero que se ejecuta aunque no lo estemos viendo es una invocación a la función `main`. Esta función contiene todo el cuerpo principal del programa.

Los parámetros (`int a`, `int b`) son variables que solo existen dentro de una función, por lo tanto si queremos acceder al valor de retorno de la función tendríamos que mandar un `return`.

Los lenguajes de hoy en día son modulares, es decir, podemos crear un conjunto de funciones y compartirlo. En Java podemos compartir un .jar, en Go un módulo, en Python un módulo a través del repositorio PayPay, ... y otros programadores podrían importarlo y utilizar nuestras funciones.

Podemos importar una API que ha liberado un tercero y utilizarlo en nuestro código. Los archivos .dll en Windows o archivos .so en Linux son librerías de funciones que las podemos invocar según nos convenga.

El cuerpo de la función es donde implementa su lógica. Las funciones deben ser útiles y pequeñas, cuanto más pequeñas son más claras.

Cuando tenemos dos variables con el mismo nombre en distintas funciones no se solapan porque están en distintos ámbitos.

A las funciones podemos invocarlas pasándole parámetros de dos formas: **paso por valor y paso por referencia**. Java no tiene el concepto de puntero, lo esconde para que sea más sencillo de utilizar.

Paso por valor. Se hace una copia del valor de una variable a otra.

Ejemplo

```
public class main {  
  
    public static void main(String[] args){  
        int param1 = 10; // 4 bytes  
        int param2 = 30; // 4 bytes  
        suma(param1, param2);  
    }  
    /* Cada vez que invocamos a la función se crea una copia de los valores y el tipo de  
dato que hemos pasado como parámetros */  
    public static void suma(int a, int b) {  
        System.out.println(a + b);  
        System.out.println(a - b);  
        System.out.println(a * b);  
        System.out.println(a / b);  
    }  
}
```

NOTA Al crear una copia de los valores y el tipo de dato, se duplica el uso de memoria. Si alguna vez, trabajamos con bases de datos y hacemos una consulta que devuelva un millón de resultados, al hacer uso del paso por valor duplicamos el uso de memoria para ese millón de resultados.

Ejemplo

```
public class pasoValor {  
  
    public static void main(String[] args) {  
        int param1 = 10;  
        int param2 = 30;  
        /* Cuando declaramos e inicializamos una variable con var el compilador  
decide de qué tipo de dato puede ser, unas veces lo hace bien y otras  
lo hace mal */  
        var valor = suma(param1, param2);  
        System.out.println(valor);  
    }  
    public static int suma(int a, int b) {  
        return a + b;  
    }  
}
```

Paso por referencia. Le pasamos una dirección (o referencia) de memoria como parámetro y la función modifica lo que haya en esa dirección de memoria. No copia los valores ni crea nuevas variables en otra zona de la memoria.

En Java es difícil verlo, la forma más efectiva de ver un paso por referencia es utilizando una instancia'.

Ejemplo

```
public class pasoReferencia {  
  
    public static void main(String[] args) {  
        /* Nombre de la clase seguido del nombre de la variable y creamos un nuevo objeto  
        Potato() */  
        Potato miPotato = new Potato();  
        miPotato.QuitarBrazo();  
        miPotato.QuitarBrazo();  
        miPotato.QuitarBrazo();  
        System.out.println(miPotato.brazos);  
    }  
    public static int suma(int a, int b) {  
        return a + b;  
    }  
}  
  
class Potato {  
    public int brazos = 0;  
    public void QuitarBrazo(){  
        this.brazos--;  
    }  
}
```

Nos da como resultado -3, hemos modificado contenido de la variable brazos de forma indirecta, con una referencia.

Hemos creado un objeto `new Potato()`; pero realmente es un puntero.

Hemos invocado a la función `miPotato.QuitarBrazo()`; tres veces y como habíamos definido el valor de brazos en cero (`brazos = 0;`) le hemos restado 3.

Y por último hemos imprimido el resultado `System.out.println(miPotato.brazos);` que nos ha dado "-3".

En todo el proceso no hemos pasado ningún parámetro y hemos decrementado la propiedad. Porque al crear un objeto le pasamos una dirección de memoria y esta es la que se modifica.

El **paso por referencia** tiene una gran ventaja en términos de optimización, si cuando **pasamos por valor** duplicamos el valor y el tipo de dato también aumentamos el consumo de memoria, en un **paso por referencia** no, porque le decimos que manipule lo que haya en la dirección de memoria.

Los objetos trabajan por debajo con punteros. Un **puntero** es una referencia a un área de la memoria del ordenador.

Cuando trabajamos habitualmente con funciones en lenguajes de programación orientados a objetos, aunque no lo sepamos se está trabajando con **punteros** (con pasos por referencia) y otras veces se trabaja con valores.

Funciones recursivas, son funciones que devuelven un valor y que se invocan a sí mismas.

Ejemplo

```
public class funcRecursiva {  
  
    public static void main(String[] args) {  
        suma(1, 2);  
    }  
    public static int suma(int a, int b) {  
        int resultado = a + b;  
        System.out.println(resultado);  
        // !!! CUIDADO !!! Esto genera un bucle infinito  
        return suma(a, resultado);  
    }  
}
```

Funciones Callback, es asignar el nombre de una función a una variable, y llamar indirectamente a la función a través de la variable. Las **CallBack** en Java no existen pero podemos pseudo implementarlas.

Ejemplo

```
/* La función funcion(); tiene que existir, y de esta forma asignamos esta función al nombre  
de la variable mivariable */  
mivariable = funcion(); // también se puede mivariable = funcion;  
// Ahora invoco a mi variable que es una función  
mivariable();  
// !!! OJO !!! Por debajo hay punteros
```

4. SENTENCIAS DE CONTROL

Las **sentencias de control** (o instrucciones) nos sirven como reglas que definimos para que un programa se comporte de una forma u otra. En las sentencias de control nos encontramos **condiciones**. Si se cumple una condición hace una cosa y si no se cumple hace otra.

Enlace a [tablas de verdad](#).

Condiciones lógicas

“Y” (dos o más cosas deben cumplirse) y se representa &&

“O” (una o ninguna o más deben cumplirse) y se representa ||

Condiciones comparativas

“MAYOR QUE”	>
“MENOR QUE”	<
“MAYOR O IGUAL QUE”	>=
“MENOR O IGUAL QUE”	<=
“IGUAL A”	==
“DESIGUAL”	!=

Ejemplo

40 MAYOR QUE 30 => CIERTO

40 MENOR QUE 30 => FALSO

(40 MAYOR QUE 30) Y (30 MENOR QUE 50)

^^^^^^^^^^^^^^^^		^^^^^^^^^^^^^^^^
SI	Y	SI
	SI	

(40 MAYOR QUE 30) O (30 MENOR QUE 50)

^^^^^^^^^^^^^^^^		^^^^^^^^^^^^^^^^
SI	O	SI
	SI	

Declarando condiciones (con pseudocódigo)

```
var estacion = "verano"
si estacion igual a "verano" entonces
    aqui las acciones a tomar
    beber_agua()
    bañarse_en_la_playa()
    tomarse_un_mojito()
    irse_al_bar()
// si la variable estacion es igual a verano se ejecuta el cuerpo de la condición
- - - - -
var estacion = "invierno"
si estacion igual a "verano" entonces
    aqui las acciones a tomar
    beber_agua()
    bañarse_en_la_playa()
    tomarse_un_mojito()
    irse_al_bar()
// si la variable estacion es diferente a verano no se ejecuta el cuerpo de la condición
- - - - -
var estacion = "verano"
var temperatura = 19
si (estacion igual a "verano") y (temperatura mayor que 20) entonces
    aqui las acciones a tomar
    beber_agua()
    bañarse_en_la_playa()
    tomarse_un_mojito()
    irse_al_bar()
/* esta condición tampoco funcionaría porque se cumple la primera condición, pero no la
segunda */
- - - - -
var estacion = "verano"
si (estacion igual a "verano") entonces
    aqui las acciones a tomar
    beber_agua()
    bañarse_en_la_playa()
    tomarse_un_mojito()
    irse_al_bar()
en caso contrario entonces:
    ponerse_el_abrigo()
    beber_chocolate_caliente()
    ver_la_tele()
/* en caso de que nuestra variable sea verano se ejecuta la primera parte de la condición, y
en caso de que nuestra variable no sea verano se ejecuta la segunda parte de la condición */
- - - - -
```

```

var estacion = "invierno"
si (estacion igual a "verano") entonces
    aqui las acciones a tomar
    beber_agua()
    bañarse_en_la_playa()
    tomarse_un_mojito()
    irse_al_bar()
en caso contrario si (estacion igual a "primavera") entonces

    salir_de_paseo()
    ir_a_ver_a_los_amigos_con_mascarilla()
en caso contrario entonces:
    ponerse_el_abrigo()
    beber_chocolate_caliente()
    ver_la_tele()
/* como nuestra variable es diferente a "verano" y "primavera" no se ejecutará ninguna de las
dos primeras condiciones, y pasará a ejecutar la tercera condición (o fallback que sería como
la acción por defecto, ya que no realizará ninguna de las acciones anteriores) */
-----

```

Ahora pasamos el pseudocódigo a sintaxis de Java:

```

public class Main {

    public static void main(String[] args) {
        String estacion = "primavera";

        if (estacion == "primavera") {
            System.out.println("Es primavera");
        } else if (estacion == "verano") {
            System.out.println("Es verano");
        } else {
            System.out.println("Es otra estacion...");
        }
    }
}

```

No es necesario que tengamos las tres condiciones, **podríamos tener un solo "if"**.

```

public class Main {

    public static void main(String[] args) {
        String estacion = "otoño";

        if (estacion == "otoño") {
            System.out.println("recogerCastañas");
        }
    }
}

```


Bucles

Un **bucle** es **hacer algo un número determinado de veces** o indeterminado, si no sabemos cuántas veces tenemos que hacerlo. Consiste en que un fragmento de código se va a ejecutar mientras se cumpla una condición. Existen **varios tipos de bucle**:

- **while**. **Evaluará la condición y si se cumple ejecutará la acción** las veces necesarias. Se suelen utilizar para comparaciones verdadero o falso.

Ejemplo

```
var contador = 10
/* el bucle while se va a ejecutar mientras se cumpla que la variable contador sea mayor que
cero */
mientras (contador mayor a cero)
    resta uno al contador
// una vez que la variable contador sea cero sigue por aquí
sigo por aquí
```

Pasamos el **pseudocódigo a lenguaje Java**:

```
public class Main {
    public static void main(String[] args) {
        int contador = 10;
        while (contador > 0) {
            System.out.println(contador);
            contador = contador - 1;
            /* la forma corta de hacer la línea anterior sería contador --; */
            /* también podríamos poner contador -= 5; para restar 5 a contador */
        }
        // por aquí seguiría el código una vez terminado el bucle
    }
}
// dejará de imprimir número en pantalla cuando la variable contador sea igual a cero
```

Una **iteración** es cuando finaliza el código del bucle y vuelve a repetirlo.

Casi todos los lenguajes de hoy en día tienen el **while** excepto Go.

- **do ... while**. **Se ejecutarán las acciones y comprobará la condición** para seguir o terminar. Como mínimo se ejecutará una vez. Muchos lenguajes no tienen el **do ... while** porque se podría construir con un **while** poniendo fuera el cuerpo.

Ejemplo

```
var contador = 10
haz
    resta uno al contador
mientras (contador sea mayor que 10)
```

Pasamos el **pseudocódigo a Java**:

```
public class Main {
    public static void main(String[] args) {
        int contador = 10;
        do {
            System.out.println(contador);
            contador = contador -1;
        } while (contador > 10);
    }
}
```

Ejemplo (construyendo con un **while** un **do ... while**)

```
public class ejemploDo_While {

    public static void main(String[] args) {
        int contador = 10;

        do {
            System.out.println(contador);
            contador = contador -1;
        }
        while (contador > 10);
    }
}
```

```
public class ejemploWhile {
    public static void main(String[] args) {
        int contador = 10;

        while (contador > 10); {
            System.out.println(contador);
            contador = contador -1;
        }
    }
}
```

- **for**. Bucle repetitivo que se **divide en 3 partes: declaración/inicialización, comparación y acción**. En algunos lenguajes estas tres partes son obligatorias y en otros no son obligatorias. Bucle muy utilizado, también con tipos de datos complejos (con listas, colecciones, sets, arrays,...).

Ejemplo

```
var contador = 10
"para" (inicializacion; comparacion; accion)
"para" ( ; contador mayor que 0; contador igual a contador menos 1)
    imprime el valor de la variable contador
/* mientras la variable contador sea mayor que cero, (primero) hará lo que tenga en el cuerpo
y (después) ejecutará la acción posterior */
```

Pasamos el **pseudocódigo a Java**:

```
public class Main {

    public static void main(String[] args) {
        for (int contador = 1; contador <= 10; contador = contador +1) {
            System.out.println(contador);
        }
        // Mi código sigue por aquí
    }
}
```

Recorriendo un **Array** en Java:

```
public class Main {

    public static void main(String[] args) {
        int valores[] = new int[5];
        /* En los bucles veremos mucho la variable i, que significa
        posición actual dentro del bucle (esto es una convención) */
        for(int i = 0; i < valores.length; i++) {
            System.out.println(valores[i]);
        }
        // Mi código sigue por aquí
    }
}
```

Las posiciones de un **Array** se empiezan a contar desde el cero. En general cuando estamos **programando empezamos a contar siempre desde cero**.

```
var valores = | 10 | 20 | 30 | 40 | 50 |
              0   1   2   3   4
```

```
para (posicion_en_array = 0; posicion_en_array < longitud_del_array; posicion_en_array++)  
    imprime el valor de la posicion actual en el array
```

Pasamos el **pseudocódigo a Java**:

```
public class Main {  
  
    public static void main(String[] args) {  
        int valores[] = {10, 20, 30, 40, 50};  
        for(int i = 0; i < valores.length; i++) {  
            // Forma corta de la línea anterior for (int i = 0 : valores) {  
                System.out.println(valores[i]);  
            }  
            // Mi código sigue por aquí  
        }  
    }  
}
```

Interruptores

Un **interrupitor** es una forma de control. Hay lenguajes que no tienen el **switch** (la traducción sería discierne). Cuando tenemos más de dos **else if** plantearnos si es mejor hacerlo con **switch**. Hay lenguajes que no tienen **switch**.

Ejemplo

```
var estacion = "verano"  
discierne (estacion)  
    caso "verano"  
        imprime "es verano"  
    caso "invierno"  
        "imprime "es invierno"  
    otro caso:  
        imprime la estacion de la variable
```

Pasamos el **pseudocódigo a Java**:

```
public class switchCase {

    public static void main(String[] args) {
        var estacion = "VERANO";

        switch (estacion) {
            case "VERANO":
                System.out.println("es verano");
                break;
            /* el break; es una palabra reservada que corta el switch al cumplirse la condición y evita
            que se sigan evaluando los casos siguientes */
            case "INVIERNO":
                System.out.println("es invierno");
                break;
            default:
                System.out.println(estacion);
                // aquí no es necesario insertar break;
        }
    }
}

-----

public class switchCase {
    public static void main(String[] args) {
        var hoy_es = "MARTES";

        switch (hoy_es) {
            case "LUNES":
            case "MARTES":
            case "MIERCOLES":
            case "JUEVES":
            case "VIERNES":
                System.out.println("hoy es laborable");
                break;
            /* En este caso evalúa el caso MARTES y como no hay break; sigue hasta el viernes, donde corta
            la ejecución del switch */
            case "SABADO":
            case "DOMINGO":
                System.out.println("hoy NO es laborable");
        }
    }
}
```

5. ERRORES

Errores a evitar y buenas prácticas a implementar:

- Evitar dar un **mal nombre a las variables**. El código debe entenderse sin necesidad de **comentarios** (esto no quiere decir que no tengamos que ponerlos), por lo que nombrar de forma correcta a las variables ayuda a que sea legible.
Utilizando bucles **for** (o **while** para contador) **no es mala práctica** utilizar variables con nombre "i", "j", ... para **usarlos como contadores**. Y **también** está **permitido** usar el nombre "temp" o "tmp" para **variables temporales**.

Ejemplo de uso de variables:

```
public class index {  
  
    public static void main(String[] args) {  
        int numeros[] = {10, 20, 30, 40, 50};  
        /* la variable "i" viene de "index" y está aceptado por la industria  
        como nombre de variable que se utiliza como un índice para recorrer el array*/  
        for (int i = 0; i < numeros.length; i++) {  
            System.out.println(numeros);  
        }  
    }  
}
```

Ejemplo de **Array** bidimensional:

```
public class arrayBid {  
  
    public static void main(String[] args) {  
        int numeros[][] = {  
            {10, 20, 30, 40, 50},  
            {10, 20, 30, 40, 50}  
        };  
        /* Recorremos el array bidimensional numeros, y por cada subarray del mismo,  
        mostramos el valor que tiene en ese momento */  
        for (int i = 0; i < numeros.length; i++) {  
            for (int j = 0; j < numeros[i].length; j++)  
                System.out.println(numeros[i][j]);  
        }  
    }  
}
```

Ejemplo de variable "temp":

```
public class Main {  
  
    public static void main(String[] args) {  
        suma(1, 2);  
    }  
  
    public static int suma(int a, int b) {  
        var temp = a + b;  
        return temp;  
    }  
}
```

- Añadir **comentarios** para aclarar para qué sirve cada bloque de código, pero **evitar** comentarios de **cosas obvias** que el código nos muestra.
- Tener en cuenta la **indentación del código** para hacerlo más legible.
En las empresas nos vamos a encontrar que tienen una **guía de estilos** y que tenemos que seguirla. Cada lenguaje de programación también tiene su propio estilo.
- Importante usar un **control de versiones** por si tenemos que revertir el código a un estado anterior. Ej.: [Github](#), [Gitlab](#), [Gitea](#), [TortoiseGit](#),...
- **No utilizar formas complejas** para hacer algo cuando haya formas más simples para hacerlo.

Ejemplo de uso de forma tradicional y forma simple:

```
public class Main {  
  
    public static void main(String[] args) {  
        int numeros[] = {10, 20, 30, 40, 50}  
    }  
  
    for (int i = 0; i < numeros.length; i++) {  
        System.out.println(numeros[i]);  
    }  
  
    // Java provee una forma más simple de hacer el for anterior y es la siguiente:  
    for (int i : numeros) {  
        System.out.println(numeros[i]);  
    }  
  
    /* En la mayoría de lenguajes de programación las variables no inicializadas se inicializan  
    con su valor 0 */  
}
```

- Cuando tenemos un problema no podemos depurar usando el **println**, los lenguajes proveen una herramienta específica llamada **depurador** para estos menesteres.
- Crear **funciones pequeñas y legibles**. No debe hacer otra cosa que no sea su cometido.
- Cuidado con la **conversión de tipos** de datos, ya que **podemos perder precisión**.

Ejemplo de **conversión de tipos** de datos (también llamado **casting**):

```
public class Main {  
  
    public static void main(String[] args) {  
        double euros = 15.900;  
        /* La función imprimeEuros(); pide un valor int y la variable euros es double,  
        hacemos una conversión de esta a formato int en la variable eurosInt. Hemos perdido  
        0.900 de precisión, ya que eurosInt devuelve 15 */  
        int eurosInt;  
        eurosInt = (int)euros; // Hacemos una conversión del tipo de dato  
        imprimeEuros(eurosInt);  
    }  
    public static void imprimeEuros(int valor) {  
        System.out.println(valor);  
    }  
}
```

- Error de **Overflow**. En los **array** debemos tener cuidado al acceder a un valor, ya que tenemos que empezar a contar desde cero.

Ejemplo de error **out of bounds (off-by-one)**:

```
public class Main {  
    public static void main(String[] args) {  
        int array[] = new int[5];  
        System.out.println(array[4]);  
        /* Se declara con el número de posiciones, pero se accede con el número de  
        posiciones -1 */  
    }  
}
```

Ejemplo de error por **overflow**:

```
public class Main {  
    public static void main(String[] args) {  
        byte numero = 127;  
        System.out.println(numero);  
        /* Con esto vamos a forzar un overflow, ya que vamos a sumar 5 a la variable  
        numero y el valor será superior al permitido para int (desde -128 al 127) */  
        numero += 5;  
        System.out.println(numero);  
    }  
}
```

- La **memoria** no es infinita. Optimizar el código y usar el tipo de dato correspondiente al uso que vamos a darle.

6. DEPURACIÓN DE CÓDIGO

La **depuración** consiste en **buscar anomalías durante la ejecución** de nuestro programa. Los depuradores tienen una ventaja y lo poseen la mayoría de los lenguajes, es lo que se conoce como **depuración remota**. Podemos ejecutar un código en un servidor de producción y desde nuestro entorno de desarrollo podemos depurar lo que está pasando en un servidor de producción.

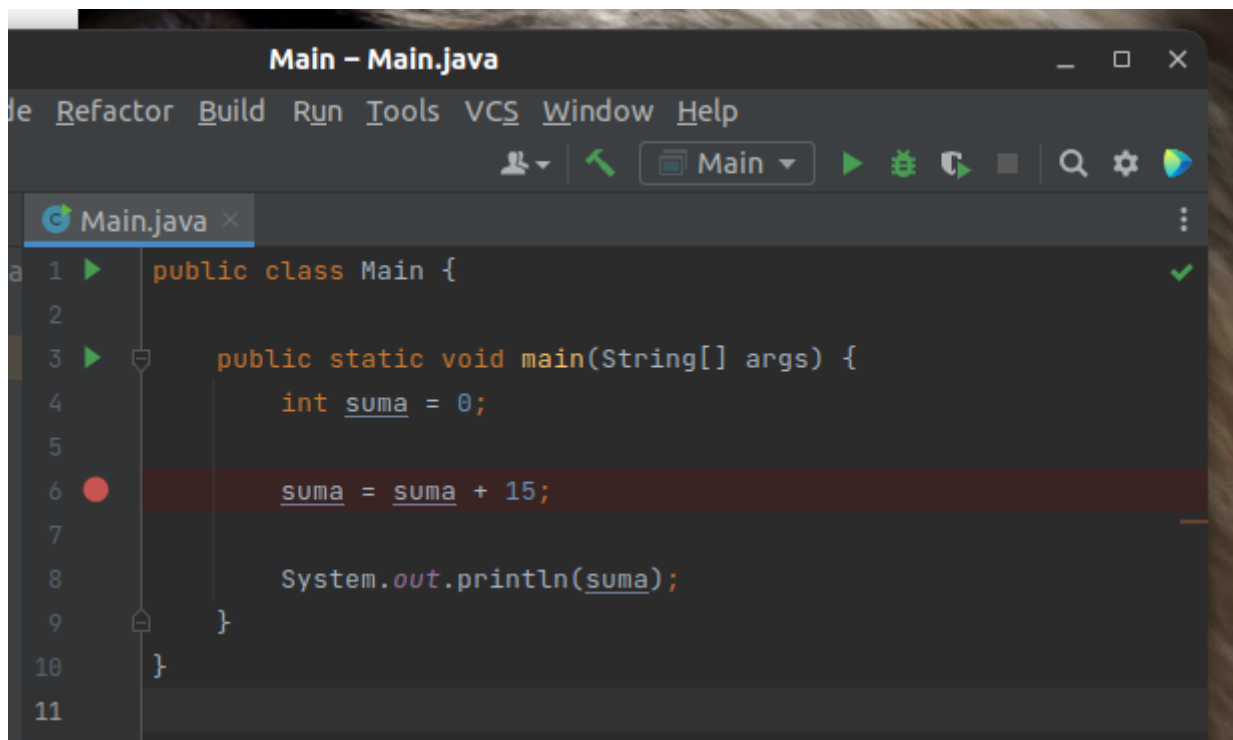
Los depuradores funcionan mediante una cosa que se llama **puntos de ruptura**. Esta es una parte en la que nosotros le decimos al programa que pare en un punto determinado de la ejecución y en ese momento salta el depurador.



Los depuradores se pueden utilizar con entorno de desarrollo o por línea de comandos (para Java tenemos **jdb**, para C tenemos **ldb** y **gdb**,...).

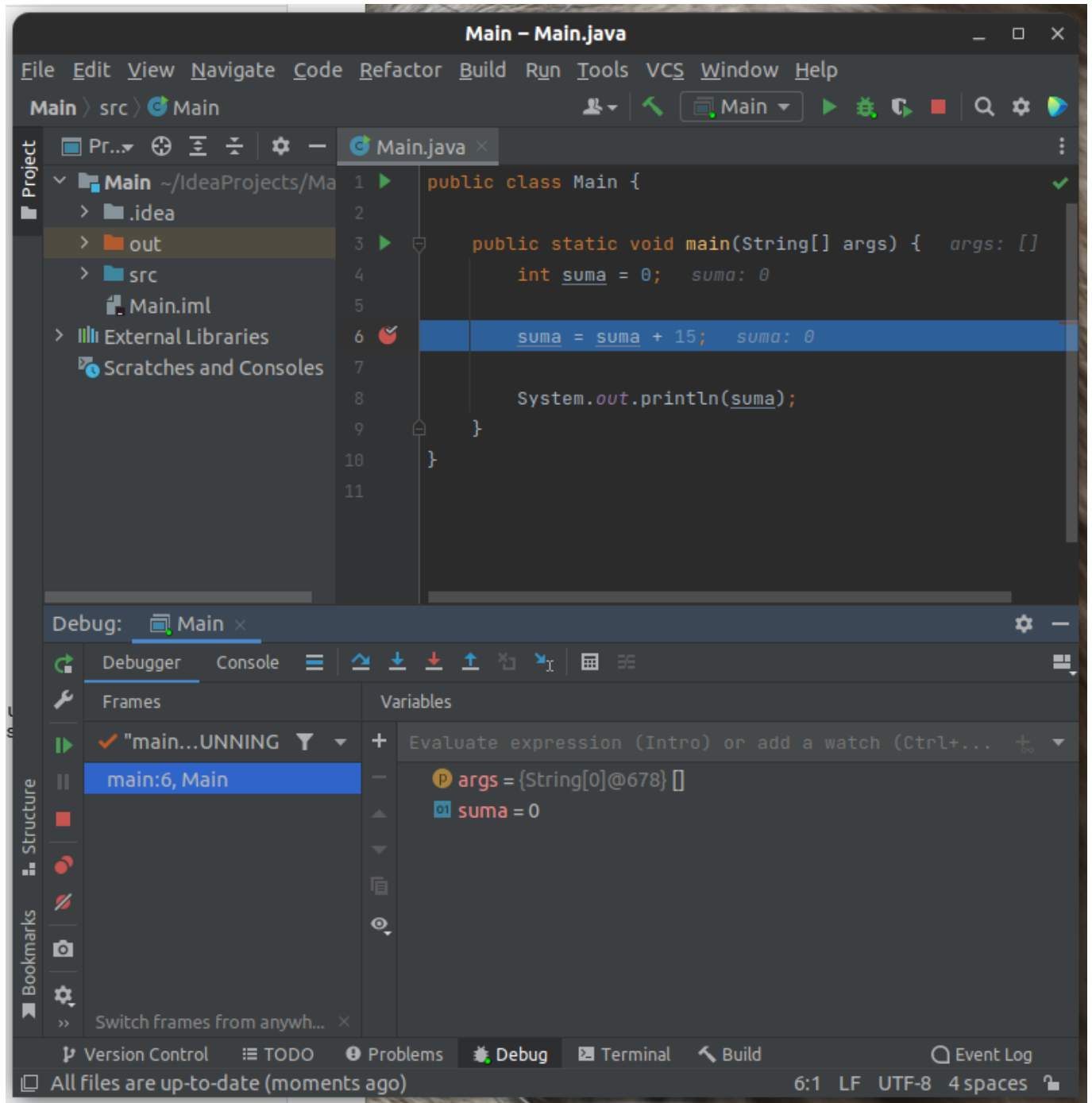
Breakpoint (o punto de ruptura)



Los puntos de ruptura se insertan en las **partes calientes de código**, es decir, las partes que suelen fallar.


En **IntelliJ** pulsamos junto al número de la línea donde queremos insertarlo y nos debe aparecer el círculo rojo ● que se ve junto a la línea 6. Ya hemos insertado el punto de ruptura.

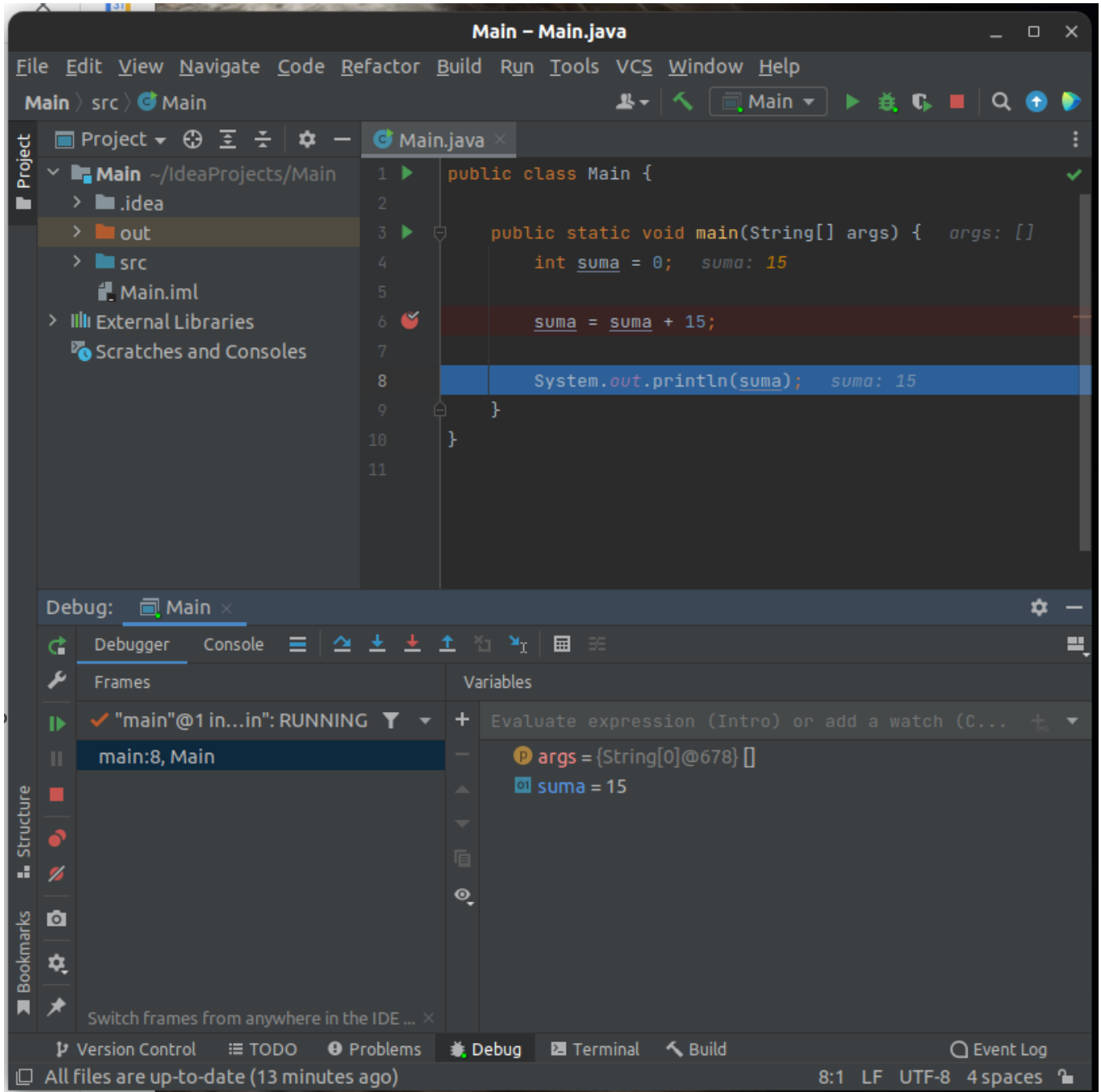


Para iniciar el **debugger** pulsamos sobre el icono  'Debug' o **Mayús + F9** y cuando el programa llega al **punto de ruptura** que hemos indicado (puntos marcados con ) el programa se detiene (pero no finaliza) para esperar a que hagamos algo. *Ver captura siguiente*

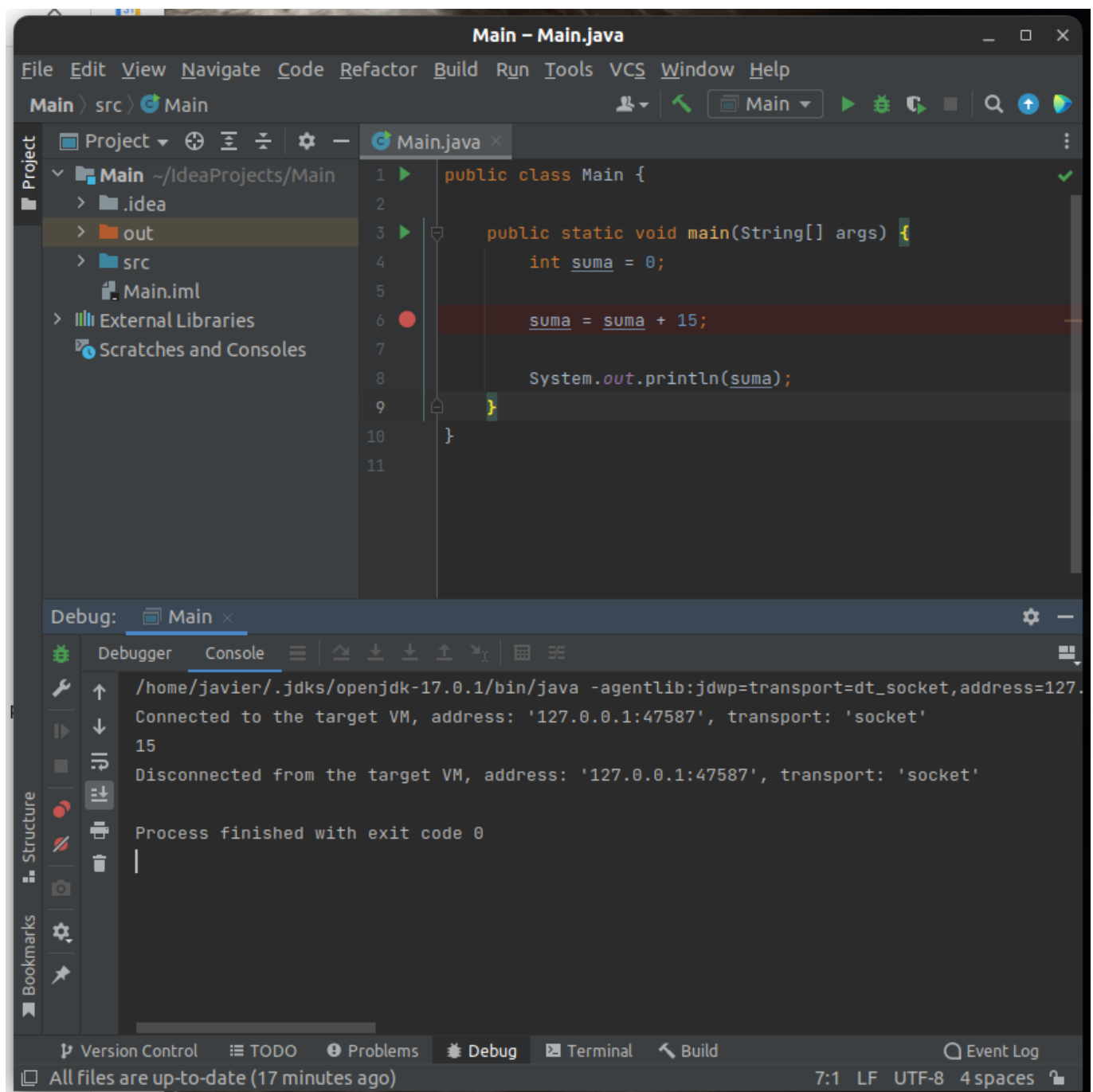


El programa se ha parado en la línea marcada con  (aparece un check  sobre este) y se ha quedado seleccionada en azul. Además en la parte inferior el panel de depuración. A la derecha nos indica que hay dos variables, la variable **suma** nos indica que **vale cero** porque se ha parado en la línea antes de ejecutarla.

Pulsamos sobre el icono  'Step Over' o F8 (que significa avanza hasta el siguiente frame) y la línea que habíamos marcado como **punto de ruptura** se queda marcada en rojo y pasa a la siguiente línea pero no la ejecuta (se queda marcada en azul).

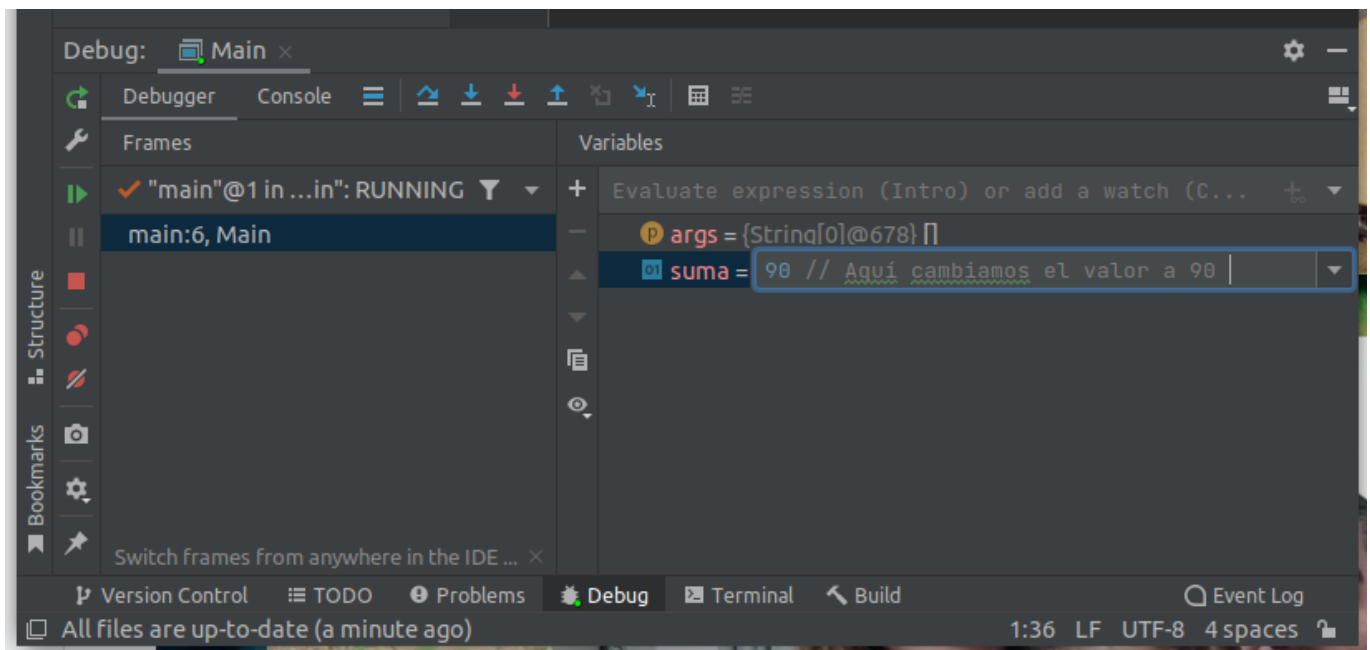


Seguimos pulsando sobre  'Step Over' o F8 y terminamos la ejecución del programa.



También **podemos alterar el valor de las variables** cuando el **depurador** llega al **punto de ruptura**. En el panel donde aparecen las variables podemos dar un valor diferente seleccionando esta y pulsando

F2 o colocando el puntero del ratón y pulsando click derecho. En el menú desplegable seleccionamos **'Set Value'**.

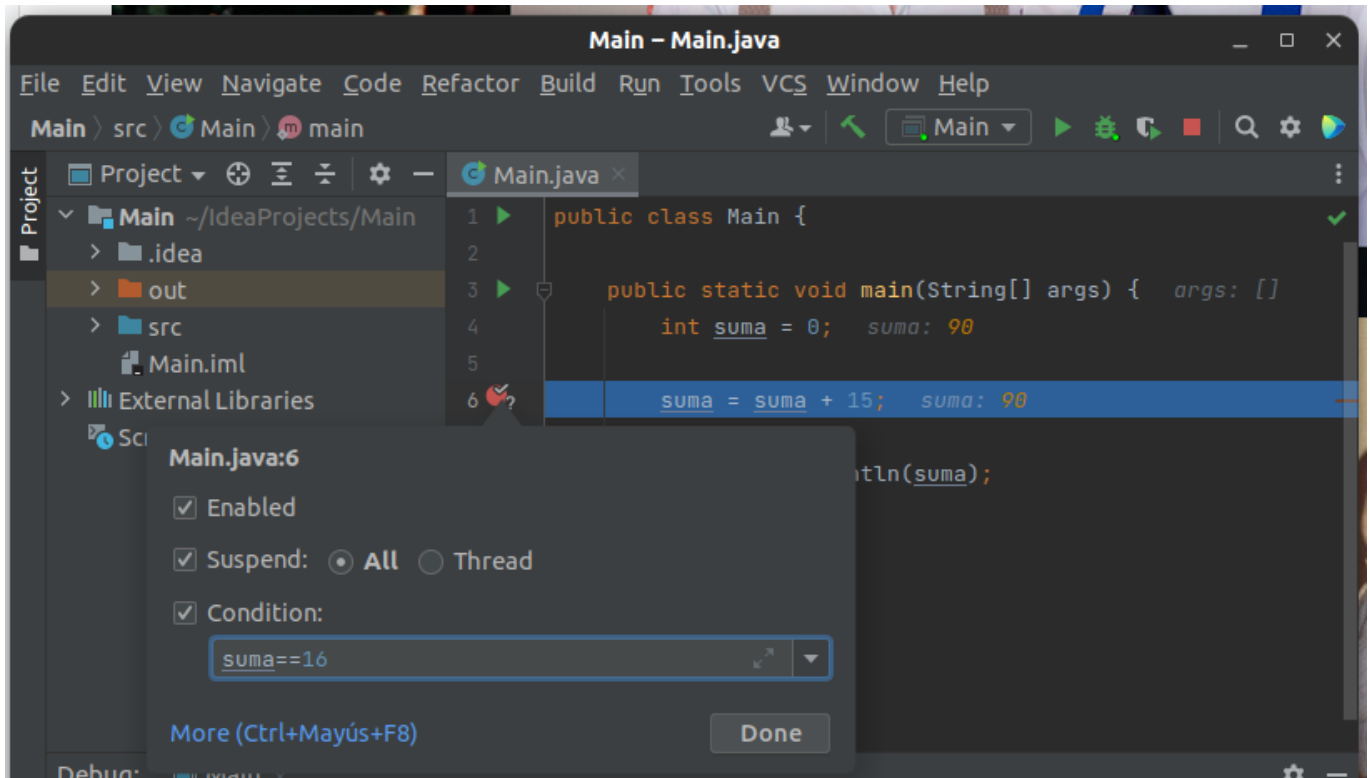


Watcher (o watch point)

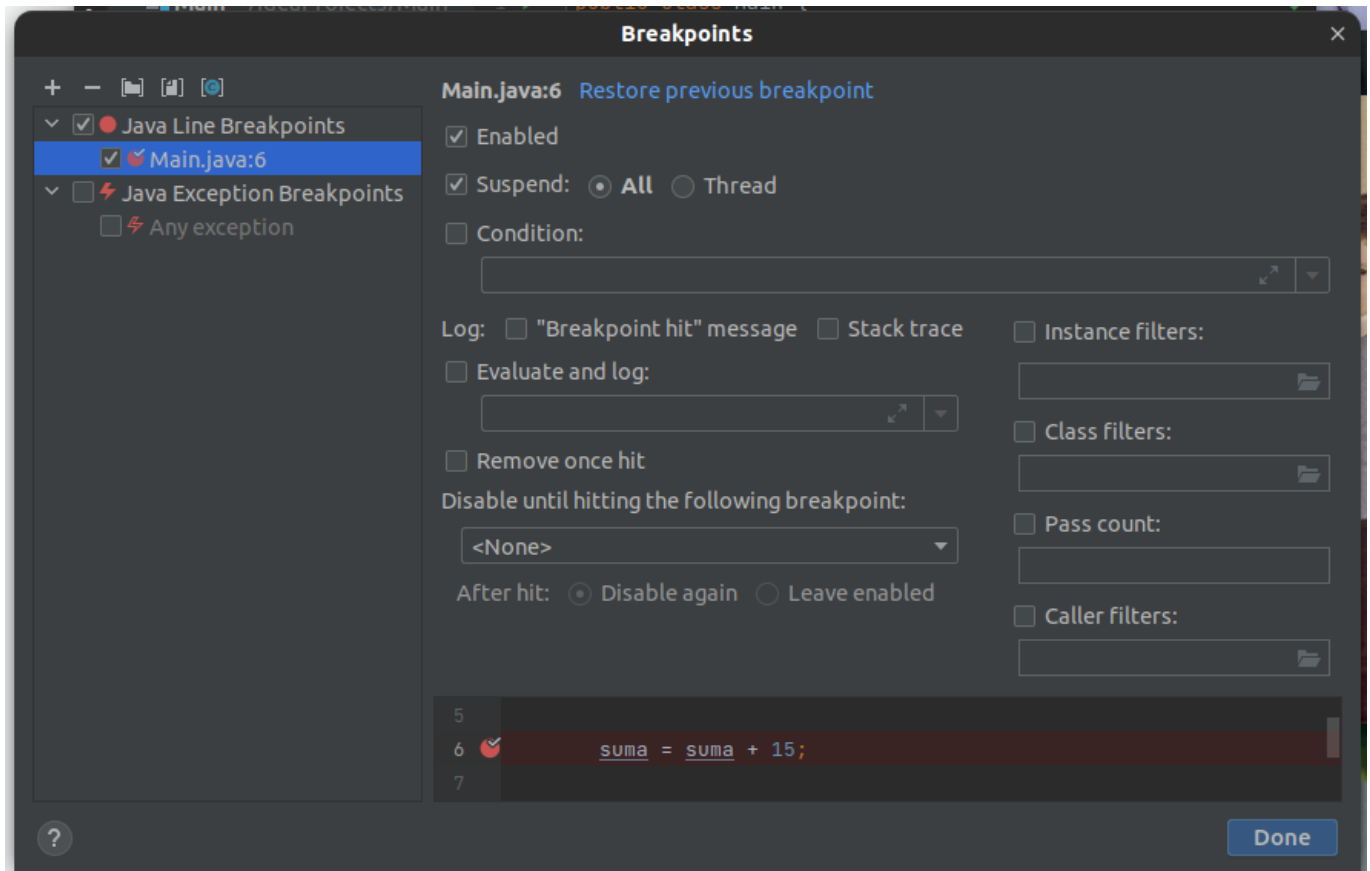
Otra opción que tenemos es que el debugger se pare cuando ocurra algo. Aquí introducimos el concepto de **watcher** que es un **breakpoint** que se dispara cuando se cumple una condición.

Para insertarlo pulsamos con el botón derecho del ratón sobre el punto rojo ● del **punto de ruptura** y nos aparece una ventana para introducir la condición.

Hemos asignado una condición para que el programa se detenga en el **breakpoint** si la variable suma vale 16. *Ver captura siguiente*

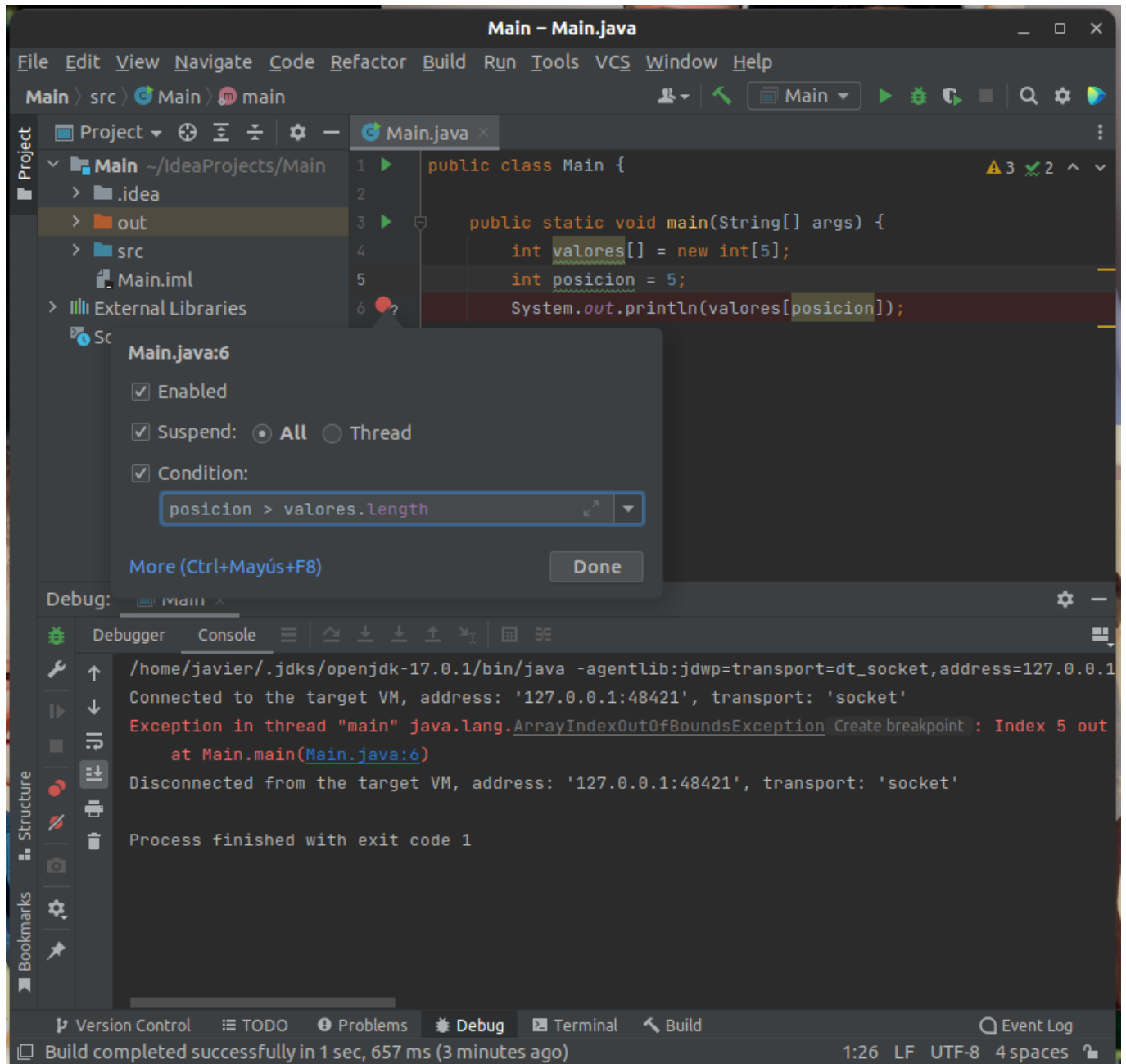


En caso de necesitar más condiciones pulsamos **More** (o **Ctrl + Mayús + F8**).



Ejemplo de uso práctico de un depurador.

Hemos creado una condición por si el valor de la posición en el array es superior al número de elementos que tiene salte el **breakpoint**.

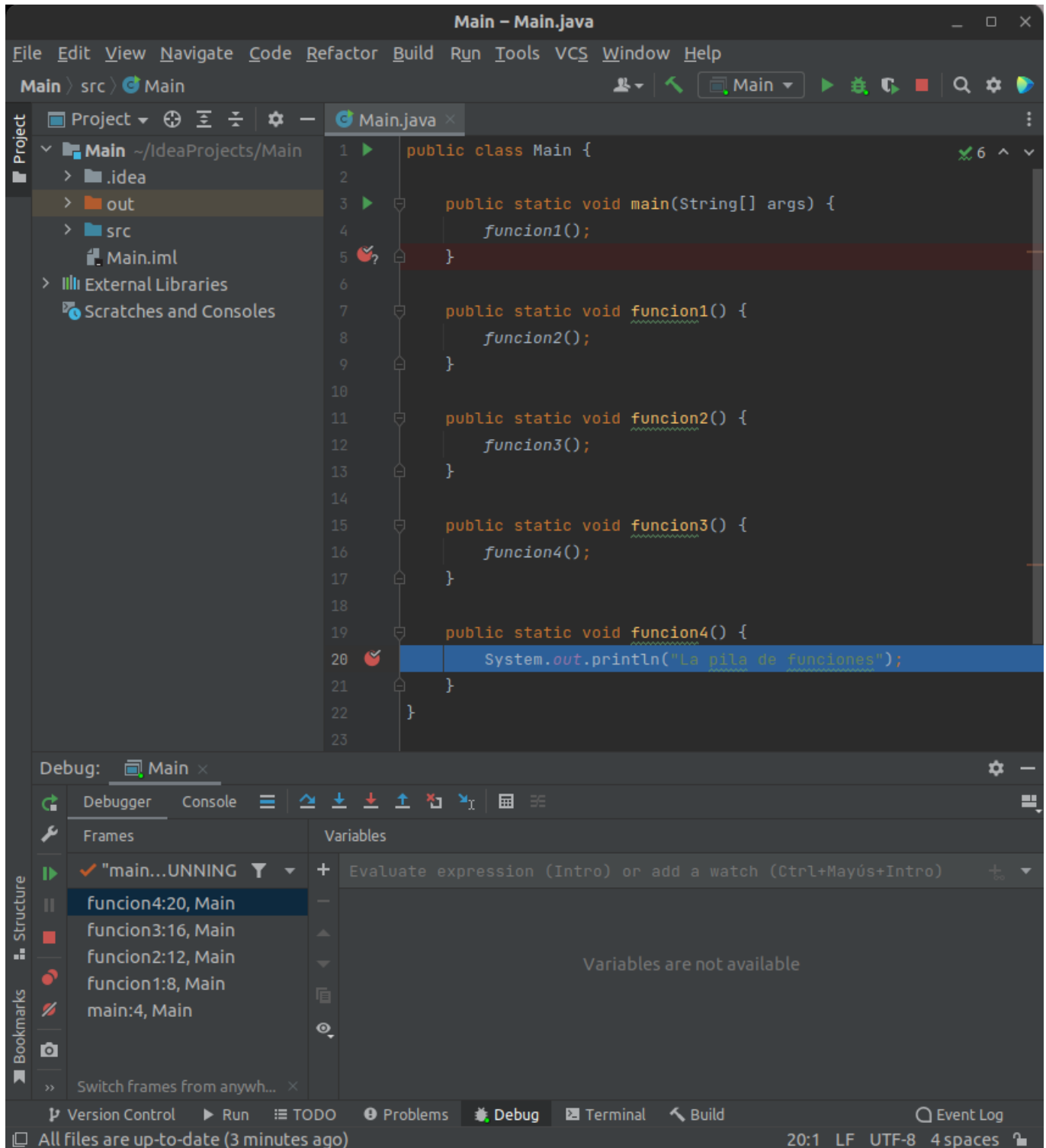


Pila de llamadas

Otra función útil es la **pila de llamadas** para saber por donde ha pasado mi programa.

Ejemplo de la pila de llamadas.

Hemos creado una serie de funciones, las cuales se van llamando unas a otras y se ha fijado un breakpoint para que cuando salte nos muestre la **pila de llamadas** en el debugger.



La **pila de llamadas** muestra cada función por la que ha pasado el programa en orden inverso, con información de la función, número de la línea de código y la clase (y también mostraría el nombre del paquete).

NOTA

Tanto **Firefox** como **Chrome** incluyen una herramienta para desarrolladores que incluye un depurador.

7. INTRODUCCIÓN A LA POO

La **POO (Programación Orientada a Objetos)** nos permite tener el código limpio y segmentado. Por ejemplo, los juegos de hoy en día están programados orientados a objetos.

Un objeto es una entidad que representa algo en el mundo real. **Para crear un objeto hacemos una instancia de una clase.** Tienen propiedades (tamaño, material,colores, peso,...) y métodos (abrir puerta, cerrar puerta,...). **Las funciones dentro de las clases se llaman métodos.**

Los objetos según el lenguaje de programación que utilicemos, se declaran mediante clases (en el caso de Java, PHP, Python, C++,...), se pueden declarar también mediante funciones dentro de estructuras (en el caso de GO).

Las clases tienen que tener una serie de **parámetros o propiedades (que son variables)** y también tiene una serie de **funciones (o métodos)**.

Para tener un objeto (en este caso **coche**) **de una clase** (en este caso **Coche**) lo hacemos instanciándolo con `new Coche()`; , lo que significa que **vamos a crear una zona de memoria a través de una variable** (en este caso **coche** -con minúscula-) y que esa zona de memoria a la que voy a acceder a través de **una variable va a tener lo que tenga la clase que hemos creado**. Es decir, vamos a crear un **objeto de una clase**, que vamos a utilizar posteriormente.

Ejemplo creando una **clase** Coche:

```
public class Main {  
  
    public static void main(String[] args) {  
    }  
}  
  
class Coche {  
    int numeroDePuertas;  
    int velocidadMaxima;  
    // Estas serían las propiedades //  
  
    public void acelerar() {}  
    public void decelerar() {}  
    // Estos serían los métodos //  
}
```

Para **crear una instancia de una clase y por tanto crear un objeto**, primero **ponemos el nombre de la clase seguido del nombre de la variable que quiero que tenga y la instanciamos así:**

```
Coche coche = new Coche();
```

Coche es un **tipo de dato** (como si fuera un int) porque es una clase.

Ejemplo creando un **objeto** de la **clase** **Coche**:

```
public class Main {  
  
    public static void main(String[] args) {  
        /* Para crear una instancia de una clase, ponemos primero el nombre de la clase Coche,  
        seguido del nombre de la variable coche. Para instanciarlo usamos new Coche(); */  
        Coche coche = new Coche();  
  
        /* Esto dice, crea una variable de nombre coche del tipo de dato Coche y a su vez la  
        inicializas new Coche(); creando una referencia nueva en memoria que sea del tipo Coche */  
  
        coche.acelerar();  
        coche.decelerar();  
    }  
}  
  
class Coche {  
    int numeroDePuertas;  
    int velocidadMaxima;  
  
    public void acelerar() {}  
    public void decelerar() {}  
}
```

A partir de ahora ya tenemos un objeto coche al cual podemos aplicarle sus funciones. Las funciones dentro de una clase las llamamos métodos.

Ejemplo aplicando métodos al objeto coche:

```
public class Main {  
  
    public static void main(String[] args) {  
        Coche coche = new Coche();  
        coche.acelerar();  
        coche.decelerar();  
    }  
}  
  
class Coche {  
    int numeroDePuertas;  
    int velocidadMaxima;  
  
    public void acelerar() {}  
    public void decelerar() {}  
}
```

Ejemplo declarando varios objetos:

```
public class Main {  
  
    public static void main(String[] args) {  
        Coche coche = new Coche();  
        System.out.println(coche.velocidadActual);  
        coche.acelerar();  
        System.out.println(coche.velocidadActual);  
  
        Coche coche2 = new Coche();  
        /* Cada uno de estos objetos (coche, coche2 y coche3) tiene las variables  
        numeroDePuertas, velocidadMaxima y velocidadActual independientes del resto de  
        coches, porque cada new Coche(); asigna una zona de memoria independiente de  
        donde estuvieran las otras variables */  
        Coche coche3 = new Coche();  
    }  
}  
  
class Coche {  
    int numeroDePuertas;  
    int velocidadMaxima;  
    float velocidadActual;  
    public void acelerar() {  
        velocidadActual += 15;  
    }  
    public void decelerar() {}  
}
```

Constructor

Un **constructor** es una forma de inicializar las propiedades de una clase cuando la instanciamos.

Cuando yo no creo un constructor, Java automáticamente lo crea por mí. Y en el caso de crearlo yo, Java no va a crearlo.

Para crear un constructor en Java tiene que seguir una serie de reglas. Lo primero es que el constructor no devuelve ningún tipo de dato (ponemos void o podemos omitirlo), el nombre tiene que ser idéntico al nombre de la clase, puede tener opcionalmente parámetros

Un constructor es lo primero que se ejecuta cuando se instancia una clase. El constructor se ejecuta en el momento que lee la línea donde declaramos el objeto coche, en este momento invoca al constructor y podemos aprovechar para inicializar ciertas variables.

Ejemplo creando **constructor** en Java:

```
public class Main {  
  
    public static void main(String[] args) {  
        Coche coche = new Coche();  
        System.out.println(coche.numeroDePuertas);  
        System.out.println(coche.velocidadActual);  
        System.out.println(coche.velocidadMaxima);  
    }  
}  
  
class Coche { // Nombre de la clase  
    int numeroDePuertas;  
    int velocidadMaxima;  
    float velocidadActual;  
  
    // Lo que sigue a continuación es cómo se define un constructor  
    public Coche() { // Nombre del constructor  
        numeroDePuertas = 5;  
        // Podemos inicializar en el constructor el valor de las variables  
        velocidadMaxima = 120;  
        System.out.println("Estoy en el constructor");  
    }  
    // Aquí termina el código del constructor  
}
```

Otro ejemplo:

```
public class Main {  
  
    public static void main(String[] args) {  
        Coche coche = new Coche(2, 90);  
        System.out.println(coche.numeroDePuertas);  
        System.out.println(coche.velocidadActual);  
        System.out.println(coche.velocidadMaxima);  
    }  
}  
  
class Coche {  
    int numeroDePuertas;  
    int velocidadMaxima;  
    float velocidadActual;  
  
    // Lo que sigue a continuación es cómo se define un constructor  
    public Coche(int puertas, int velocidad) {  
        numeroDePuertas = puertas;  
        velocidadMaxima = velocidad;  
        System.out.println("Estoy en el constructor");  
    }  
    // Aquí termina el código del constructor  
}
```

Otro ejemplo:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche(2, 90); /* Aquí creamos el objeto coche con el
        tipo de dato Coche, ya que este es una clase y lo podemos utilizar como tipo de
        dato */
        System.out.println(coche.velocidadActual);
        coche.acelerar();
        System.out.println(coche.velocidadActual);
        Coche coche2 = new Coche(3, 20);
        System.out.println(coche2.velocidadActual);
        Coche coche3 = new Coche(1, 30);
        System.out.println(coche3.numeroDePuertas);
    }
}

class Coche { //Aquí definimos la clase Coche
    int numeroDePuertas; // Propiedades de la clase
    int velocidadMaxima;
    float velocidadActual;

    // Lo que sigue a continuación es como se define un constructor
    public Coche(int puertas, int velocidad) {
        /* El constructor debe tener idéntico nombre que la clase y le estamos
        diciendo que la clase Coche inicializa el objeto coche con 2 parámetros */
        numeroDePuertas = puertas;
        velocidadMaxima = velocidad;
        System.out.println("Estoy en el constructor");
    }
    // Aquí termina el código del constructor
    public void acelerar() {
        velocidadActual += 15;
    }
    public void decelerar() {}
}
```

Sobrecarga

En Java existe el concepto de **sobrecarga**, que consiste en que **puedo tener dos funciones con el mismo prototipo pero distintos parámetros**.

Ejemplo de sobrecarga:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        System.out.println(coche.numeroDePuertas);
        System.out.println(coche.velocidadMaxima);
        System.out.println(coche.velocidadActual);
        Coche coche2 = new Coche(2, 90);
        System.out.println(coche2.numeroDePuertas);
        System.out.println(coche2.velocidadMaxima);
        System.out.println(coche2.velocidadActual);
    }
}

class Coche {
    int numeroDePuertas;
    int velocidadMaxima;
    float velocidadActual;

    // INICIO CONSTRUCTOR 1
    public Coche() {
        numeroDePuertas = 5;
        velocidadMaxima = 120;
        System.out.println("Estoy en el constructor SIN PARÁMETROS");
    }
    // FIN CONSTRUCTOR 1
    /* La diferencia entre uno y otro es que si lo invoco sin parámetros disparará el
    CONSTRUCTOR 1 y si lo invoco con parámetros dispara el CONSTRUCTOR 2 */
    // INICIO CONSTRUCTOR 2 con parámetros
    public Coche(int puertas, int velocidad) {
        numeroDePuertas = puertas;
        velocidadMaxima = velocidad;
        System.out.println("Estoy en el constructor CON PARÁMETROS");
    }
    // FIN CONSTRUCTOR 2

    public void acelerar() {
        velocidadActual += 15;
    }
    public void decelerar() {}
}
```

Esto sería una **sobrecarga del constructor o constructor overloading**.

Buena práctica

Cuando tenemos un **constructor con parámetros** y queremos inicializar **variables internas** habitualmente le vamos a dar como nombre de los parámetros los **mismos nombres de propiedades que tenga la clase**.

Pero al querer asignar el **parámetro** a la **propiedad de la clase**, estas se llamarían igual, por lo que tendríamos un conflicto de nombres.

Cuando tenemos dos variables iguales, una de ellas que pertenece a la clase y otra de ellas que es un parámetro (o argumento) de la función, para hacer referencia a la variable de la clase le anteponemos **"this."**

Ejemplo:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        System.out.println(coche.numeroDePuertas);
        System.out.println(coche.velocidadMaxima);
        Coche coche2 = new Coche(2, 90);
        System.out.println(coche2.numeroDePuertas);
        System.out.println(coche2.velocidadMaxima);
    }
}

class Coche {
    int numeroDePuertas;
    int velocidadMaxima;
    float velocidadActual;

    public Coche() { // Esto es el constructor 1
        numeroDePuertas = 5;
        velocidadMaxima = 120;
        System.out.println("Estoy en el constructor SIN PARÁMETROS");
    }

    public Coche(int numeroDePuertas, int velocidadMaxima) { // Esto es el constructor 2
        this.numeroDePuertas = numeroDePuertas;
        /* Al parámetro numeroDePuertas de esta clase asigne el valor de la variable
        de este parámetro numeroDePuertas */
        this.velocidadMaxima = velocidadMaxima;
        System.out.println("Estoy en el constructor CON PARÁMETROS");
    }
}
```

8. PRIVACIDAD, ABSTRACCIÓN Y ENCAPSULACIÓN

CLASE MICLASE

```
PROPIEDAD1;  
PROPIEDAD2;
```

```
FUNCION1();  
FUNCION2();
```

- PROGRAMA PRINCIPAL -

```
VAR unacalse INSTANCIA DE MICLASE
```

```
unacalse.PROPIEDAD1 = valor  
IMPRIME unacalse.PROPIEDAD2
```

Propiedades privadas y públicas

Las propiedades pueden ser de 3 tipos, pero por ahora veremos las **públicas** y las **privadas**.

Cuando podemos **acceder desde fuera de una clase a sus propiedades** directamente estaríamos diciendo que las **propiedades son públicas**.

Diferencia entre propiedad pública y privada. La **propiedad privada solo se puede utilizar en la implementación de la clase** y la **propiedad pública la podemos utilizar dentro de la clase y fuera de ella**.

¿**Cómo se define** cuándo es **público** y cuando es **privado**?, depende del lenguaje. En el caso de **Java** lo definimos al crear la propiedad, aunque en cualquier lenguaje que tenga un buen soporte de programación orientada a objetos lo va a implementar igual (por ejemplo, PHP lo hace igual).

Ejemplo:

CLASE MICLASE

```
public PROPIEDAD1;  
private PROPIEDAD2;  
FUNCION1();  
FUNCION2();
```

- PROGRAMA PRINCIPAL -

```
VAR unacalse INSTANCIA DE MICLASE // Creamos un objeto
```

```
unacalse.PROPIEDAD1 = valor // Accedemos a la propiedad PROPIEDAD1 que es pública  
unacalse.PROPIEDAD2 = valor // No podemos acceder a PROPIEDAD2 por ser privada  
IMPRIME unacalse.PROPIEDAD1 // Imprime valor de PROPIEDAD1 del objeto unacalse
```

Ejemplo de propiedad pública:

```
public class Main {

    public static void main(String[] args) {
        // Creamos el objeto vehiculo, que es una instancia de la clase Vehiculo
        Vehiculo vehiculo = new Vehiculo();
        // Asignamos el valor "Coche" a la propiedad tipo del objeto vehiculo
        vehiculo.tipo = "Coche";
        // Imprimimos por pantalla el valor de la propiedad tipo del objeto vehiculo
        System.out.println(vehiculo.tipo);
    }
}

class Vehiculo { // Creamos la clase Vehiculo
    public String tipo;
    private int numeroRuedas; // Esta propiedad no sería accesible desde fuera de la clase
}
```

Encapsulación (getters y setters)

La **encapsulación** consiste en **jugar con los tipos públicos y privados** de forma que desde la clase los manipule y desde fuera de la clase los pueda utilizar.

Para **encapsular** declaramos las propiedades de la clase como privadas y luego **tenemos que crear dos funciones por cada una de las propiedades** (a esto lo llamamos **getters y setters** porque estas funciones tienen como tarea modificar la propiedad o darnos el valor de la propiedad).

Se llama encapsulación porque **estamos encapsulando las propiedades para acceder a ellas únicamente a través de funciones**.

Ejemplo de **setter** y **getter** para establecer la PROPIEDAD1 al valor que queramos:

```
CLASE MICLASE
    private PROPIEDAD1;
    public PROPIEDAD2;

    FUNCION SETTERPROPIEDAD1(TEXT0 valor)
        ESTA_CLASE.PROPIEDAD1 = valor
    FUNCION GETTERPROPIEDAD1() TEXT0
        DEVUELVE EL VALOR DE ESTA_CLASE.PROPIEDAD1

- PROGRAMA PRINCIPAL -
    VAR unaclase INSTANCIA DE MICLASE // Creamos un objeto

    unaclase.PROPIEDAD1 = valor // Accedemos a la propiedad PROPIEDAD1 del objeto unaclase
    IMPRIME unaclase.PROPIEDAD2
```

Pasamos el **pseudocódigo a Java** y creamos la función `setTipo()`:

```
public class Main {  
  
    public static void main(String[] args) {  
        Vehiculo vehiculo = new Vehiculo();  
        /* Asignamos el valor "Coche" a la propiedad tipo (que es un String) del objeto vehiculo */  
        vehiculo.tipo = "Coche";  
        System.out.println(vehiculo.tipo);  
    }  
}  
  
class Vehiculo {  
    private String tipo;  
  
    /* Los setters no devuelven nada y toman como tipo de dato el tipo de dato de la propiedad */  
    public void setTipo(String tipo) {  
        // this.tipo hace referencia a la propiedad tipo de la clase  
        this.tipo = tipo;  
        // tipo hace referencia al parámetro que estoy recibiendo de la función setTipo()  
    }  
}
```

La convención en Java es que los **setters** empiezan por la palabra **set** (en minúscula) y a continuación el **nombre de la variable** capitalizando la primera letra.

Los **setters** nunca devuelven nada y toman como tipo de dato para el parámetro el tipo de dato de la propiedad (en este caso es un `String`). Además hacemos coincidir el nombre de la variable con el nombre de la propiedad.

Los **getters** a diferencia de los **setters** no fijan el valor, sino que lo devuelve.

Cuando creamos la función **get**, como obtiene información y nos la va a devolver, tiene que tener un tipo de retorno, y el tipo de retorno que vamos a utilizar es el tipo de retorno del tipo de dato que vamos a devolver. En este ejemplo `tipo` es `String`, vamos a devolver un `String`.

La implementación del **getter** es prácticamente igual al **setter**, pero no tiene parámetros porque no modifica nada.

Ejemplo con las funciones **setTipo** y **getTipo**:

```
public class Main {
    public static void main(String[] args) {
        Vehiculo vehiculo = new Vehiculo();
        vehiculo.setTipo("Coche");

        String tipo = vehiculo.getTipo();
        System.out.println(tipo);
        // Las dos líneas anteriores se pueden sustituir por la siguiente:
        // System.out.println(vehiculo.getTipo());
    }
}

class Vehiculo {
    private String tipo;

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public String getTipo() {
        return this.tipo;
    }
}
```

Ejemplo

```
public class Main {

    public static void main(String[] args) {
        Vehiculo coche = new Vehiculo();
        coche.setTipo("Coupe");

        Vehiculo moto = new Vehiculo();
        moto.setTipo("Scotter");

        System.out.println(coche.getTipo());
        System.out.println(moto.getTipo());
    }
}

class Vehiculo {

    private String tipo;

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public String getTipo() {
        return this.tipo;
    }
}
```

Ejemplo

```
public class Main {

    public static void main(String[] args) {
        Vehiculo coche = new Vehiculo();
        coche.setTipo("Coupe");
        coche.setVelocidadMaxima(120);

        Vehiculo moto = new Vehiculo();
        moto.setTipo("Scotter");
        moto.setVelocidadMaxima(50);

        System.out.println(coche.getTipo());
        System.out.println(coche.getVelocidadMaxima());
        System.out.println(moto.getTipo() + " " + moto.getVelocidadMaxima());
    }
}

class Vehiculo {

    private String tipo;
    private int velocidadMaxima;

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
    public String getTipo() {
        return this.tipo;
    }
    public void setVelocidadMaxima(int velocidadMaxima){
        this.velocidadMaxima = velocidadMaxima;
    }
    public int getVelocidadMaxima() {
        return this.velocidadMaxima;
    }
}
```

Hay un **setter y getter especiales** para los tipos de datos **boolean**. Cuando tenemos un tipo de dato boolean el setter es igual, pero **la función que haría de getter se formaría anteponiendo la palabra **is****.

Ejemplo de **getter** y **setter** para tipo de **dato boolean**:

```
public class Main {

    public static void main(String[] args) {
        Vehiculo coche = new Vehiculo();
        coche.setTipo("Coupe");
        coche.setVelocidadMaxima(120);
        coche.setRapido(true);

        Vehiculo moto = new Vehiculo();
        moto.setTipo("Scotter");
        moto.setVelocidadMaxima(50);
        moto.setRapido(false);

        System.out.println(coche.getTipo());
        System.out.println(coche.getVelocidadMaxima());
        System.out.println(coche.isRapido());
    }
}

class Vehiculo {

    private String tipo;
    private int velocidadMaxima;
    private boolean rapido;

    public void setRapido(boolean rapido) {
        this.rapido = rapido;
    }
    public boolean isRapido() { // La función get cambia para un tipo de dato boolean
        return this.rapido;
    }
    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
    public String getTipo() {
        return this.tipo;
    }
    public void setVelocidadMaxima(int velocidadMaxima){
        this.velocidadMaxima = velocidadMaxima;
    }
    public int getVelocidadMaxima() {
        return this.velocidadMaxima;
    }
}
```

Abstracción

La **abstracción** consiste en implementar parte de mi clase y dejar parte de mi clase a su libre albedrío. Una **clase abstracta o un método abstracto** (función dentro de una clase) es igual que una clase normal y corriente, pero suele tener un calificador.

Ejemplo en pseudocódigo:

```
CLASE ABSTRACTA VEHICULO
    PRIVADA TIPO;
    PRIVADA SONIDO;

    FUNCION SETTERTIPO(TEXT0 valor)
        ESTA_CLASE.TIPO = valor
    FUNCION ABSTRACTA GETTERSONIDO() TEXT0
        DEVUELVE "ALGO"

- PROGRAMA PRINCIPAL -
    VAR unacalse INSTANCIA DE MICLASE

    unacalse.PROPIEDAD1 = valor
    IMPRIME unacalse.PROPIEDAD2
```

Una **función abstracta** quiere decir que el lenguaje subyacente (Java, C#, ...) puede implementar el código de algunas funciones y pueden decirnos que la función abstracta la tiene que implementar una **clase hija**. Es decir, implementan la funcionalidad de forma parcial y es el programador el que posteriormente tendrá que implementar el resto de funciones, nos darán unas sí, otras no.

La **clase abstracta** no la podemos utilizar directamente, es decir, no las podemos instanciar, tienen que ser **heredadas**.

La **clase abstracta** es una clase normal y corriente en la cual nos dan una serie de funciones ya programadas y nos dicen que tenemos que programar otra serie de funciones nosotros.

Ejemplo en pseudocódigo:

```
CLASE ABSTRACTA VEHICULO
    PRIVADA TIPO;
    PRIVADA SONIDO;

    FUNCION SETTERTIPO(TEXT0 valor)
        ESTA_CLASE.TIPO = valor

    FUNCION GETTERTIPO() TEXT0
        DEVUELVE ESTA_CLASE.TIPO
    // Estas dos funciones anteriores nos las darán hechas

    FUNCION ABSTRACTA SETTERSONIDO(TEXT0 sonido)

    FUNCION ABSTRACTA GETTERSONIDO() TEXT0
    // Estas dos funciones anteriores NO nos las darán hechas

CLASE COCHE
    FUNCION SETTERSONIDO(TEXT0 sonido)
    FUNCION GETTERSONIDO() TEXT0

- PROGRAMA PRINCIPAL -
    VAR unacalse INSTANCIA DE MICLASE

    unacalse.PROPIEDAD1 = valor
    IMPRIME unacalse.PROPIEDAD2
```

Con el ejemplo anterior veríamos que es una **clase abstracta** y tenemos que saber que nos van a dar una serie de funciones (o métodos, porque están dentro de una clase) ya hechas, y otras no están hechas. Nos dirán que cuando queramos utilizar la clase VEHICULO vamos a tener que **derivarla** (este concepto se verá en el siguiente punto **HERENCIA**) y una vez que la derivemos, en mi nueva clase tengo que implementar obligatoriamente las funciones abstractas.

Una **función abstracta** **no puede tener un body**, es decir, a diferencia de las funciones normales **son prototipos**.

Las **clases abstractas** se llaman también **clases parciales**, ya que nos dan una parte hecha y otra no.

Ejemplo en Java:

```
public class Main {  
  
    public static void main(String[] args) {  
    }  
}  
  
abstract class Vehiculo {  
  
    private String tipo;  
    private int velocidadMaxima;  
    private String sonido;  
  
    abstract public void setSonido(String sonido);  
    abstract public String getSonido();  
  
    public void setVelocidadMaxima(int velocidadMaxima){  
        this.velocidadMaxima = velocidadMaxima;  
    }  
    public int getVelocidadMaxima() {  
        return this.velocidadMaxima;  
    }  
    public void setTipo(String tipo) {  
        this.tipo = tipo;  
    }  
    public String getTipo(){  
        return this.tipo;  
    }  
}
```

Con el ejemplo anterior, nos van a dar las funciones setter y getter ya implementadas (para las propiedades `velocidadMaxima` y `tipo`), pero nosotros vamos a tener que programar la función `setSonido` como `getSonido`.

También podríamos crear una librería con la clase abstracta Vehiculo y pasarla a un programador, y ya le estaríamos dando una serie de métodos.

9. HERENCIA, POLIMORFISMO E INTERFACES

Herencia

La **herencia** consiste en que **una clase hereda métodos y propiedades de otra clase**. A la clase que hereda la llamamos **clase hija** y a la clase que cede sus funciones la llamamos como queramos (superClase, claseBase, clasePrincipal...).

Cada **clase derivada** hereda las propiedades y métodos de su padre, y añade las suyas propias.

Ejemplo en pseudocódigo:

```
CLASE VEHICULO
    PRIVADA VELOCIDADMAXIMA;
    PRIVADA TIPOGASOLINA;

    FUNCION diHola()
        IMPRIME "Hola"

CLASE COCHE HEREDA DE VEHICULO;
    (heredada) VELOCIDADMAXIMA
    (heredada) TIPOGASOLINA
    NUMERO_DE_PUERTAS;

    (heredada) FUNCION diHola()
        IMPRIME "Hola"

    FUNCION SETTERNUMERODEPUERTAS(INTEGER puertas)
        ESTA_CLASE.NUMERO_DE_PUERTAS = puertas

    FUNCION GETTERNUMERODEPUERTAS() INTEGER
        DEVUELVE ESTA_CLASE.NUMERO_DE_PUERTAS

CLASE COUPE HEREDA DE COCHE;
    (heredada) VELOCIDADMAXIMA
    (heredada) TIPOGASOLINA
    (heredada) NUMERO_DE_PUERTAS;

    (heredada) FUNCION diHola()
        IMPRIME "Hola"

    (heredada) FUNCION SETTERNUMERODEPUERTAS(INTEGER puertas)
        ESTA_CLASE.NUMERO_DE_PUERTAS = puertas

    (heredada) FUNCION GETTERNUMERODEPUERTAS() INTEGER
        DEVUELVE ESTA_CLASE.NUMERO_DE_PUERTAS
```

Herencia múltiple hace referencia a la característica de los lenguajes de programación orientada a objetos en la que una clase puede heredar comportamientos y características de más de una superclase. Esto contrasta con la **herencia simple**, donde una clase solo puede heredar de una superclase.

Ejemplo creando una clase derivada en Java:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.velocidadMaxima = 14;
        coche.matricula = " AAA 1234";

        CocheElectrico cocheElectrico = new CocheElectrico();
        cocheElectrico.velocidadMaxima = 10;
        cocheElectrico.matricula = "BBB 4321";
    }
}

class Vehiculo {
    int velocidadMaxima;
    String matricula;
}

class Coche extends Vehiculo {}
/* Hemos declarado una clase hija Coche derivada de la clase padre Vehiculo, y para ello
usamos extends */

final class CocheElectrico extends Coche {}
/* Hemos declarado una clase hija CocheElectrico derivada de la clase padre Coche, pero al
declararla como final no se podrán heredar */
```

Hasta un tercer o cuarto nivel de herencia estaría bien, no derivar más de esto.

También tenemos la opción de no permitir que puedan crear clases hijas de una clase. Se hace anteponiendo **final** a la clase.

Ejemplo otra más:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.velocidadMaxima = 14;
        coche.matricula = "AAA 1234";

        CocheElectrico cocheElectrico = new CocheElectrico();
        cocheElectrico.velocidadMaxima = 10;
        cocheElectrico.matricula = "BBB 4321";

        System.out.println(cocheElectrico.compruebaMatricula("AAA"));
    }
}

class Vehiculo {
    int velocidadMaxima;
    String matricula;

    public boolean compruebaMatricula(String matricula) {
        if (matricula == "AAA") {
            return true;
        }

        return false;
    }
}

class Coche extends Vehiculo {}

final class CocheElectrico extends Coche {}
```

¿En qué casos puedo utilizar una **clase padre** o una **clase hija**? Nunca voy a utilizar una clase padre cuando dependa de un método de una clase hija. Si la clase hija implementa métodos que no están en la clase padre, desde esta no tendríamos acceso a los métodos de la clase hija o sus derivadas.

Cuando tenemos una clase abstracta con métodos abstractos y esta la derivamos, tenemos que implementar en la clase hija los métodos marcados como abstractos en la clase padre.

NOTA

Cuando tenemos una clase abstracta significa que no podemos crear instancias de esta clase, pero si podemos crear clases hijas. Y dentro de una clase abstracta, puedo tener métodos abstractos (nos dan el nombre del método pero tenemos que implementarla en la clase hija) o no, pero en el caso de declarar un método como abstracto la clase debe ser declarada abstracta.

Ejemplo con clases abstractas, heredadas y métodos abstractos:

```
public class Main {
    public static void main(String[] args) {
        Coche coche= new Coche();
        coche.setSonido("BRRR");
        System.out.println(coche.getSonido());
    }
}
abstract class Vehiculo {
    int velocidadMaxima;
    String matricula;
    String sonido;

    public Vehiculo() {
        System.out.println("Estoy en el constructor de Vehiculo");
    }
    abstract public String getSonido();
    abstract public void setSonido(String sonido);
}
class Coche extends Vehiculo {
    public String getSonido() {
        return "Soy un supersonido: " + this.sonido;
    }
    public void setSonido(String sonido) {
        this.sonido = sonido;
    }
}
class Moto extends Vehiculo {

    public String getSonido() {
        return "Soy un sonidillo de moto: " + this.sonido;
    }
    public void setSonido(String sonido) {
        this.sonido = sonido;
    }
}
```

Hay diferentes **tipos de herencia**, es decir, **las relaciones que creamos entre las clases**. No es que haya diferentes formas de decirlo, al final son herencias simples, una clase hereda de otra.

La **herencia multinivel** es cuando una clase hereda de una clase que a su vez ha heredado de otra y así sucesivamente. Y la **herencia simple** es cuando una clase ha heredado de otra.

Ejemplo de herencia múltiple en **pseudocódigo**:

```
CLASE VEHICULO
PRIVADA VELOCIDADMAXIMA;
    PRIVADA TIPOGASOLINA;

    FUNCION diHola()
        IMPRIME "Hola"

CLASE MOTOR
    PRIVADA TIPOGASOLINA;

CLASE COCHE HEREDA DE VEHICULO Y DE MOTOR;

    (heredada) VELOCIDADMAXIMA
    (heredada) TIPOGASOLINA
    NUMERO_DE_PUERTAS;

    (heredada) FUNCION diHola()
        IMPRIME "Hola"

    FUNCION SETTERNUMERODEPUERTAS(INTEGER puertas)
        ESTA_CLASE.NUMERO_DE_PUERTAS = puertas

    FUNCION GETTERNUMERODEPUERTAS() INTEGER
        DEVUELVE ESTA_CLASE.NUMERO_DE_PUERTAS
```

Ejemplo pasándolo a **Java**:

```
public class Main {

    public static void main(String[] args) {
        Coche coche= new Coche();
        coche.setSonido("BRRR");
        System.out.println(coche.getSonido());
    }
}

class Vehiculo {
    int velocidadMaxima;

    public Vehiculo() {
        System.out.println("Estoy en el constructor de Vehiculo");
    }
}

class Motor {
    String tipomotor;
    public Motor() {
        System.out.println("Estoy en el constructor de Motor");
    }
}

// La clase Coche va a heredar de Vehiculo y Motor, esto es una herencia múltiple.
class Coche extends Vehiculo, Motor {
    public Coche() {
    }
}
```

La **herencia jerárquica** consiste en que de una clase base derivan otras clases, por ejemplo, de una clase padre derivan dos clases hijas y a su vez de cada una de estas dos clases hijas derivan otras dos clases hijas. Sería como un árbol genealógico.

Ejemplo de herencia jerárquica:

```
CLASE A
  CLASE B HEREDA DE A
    CLASE UNO HEREDA DE B
    CLASE DOS HEREDA DE B
  CLASE C HEREDA DE A
    CLASE PERRO HEREDA DE C
    CLASE GATO HEREDA DE C
  CLASE D HEREDA DE C
    CLASE COCHE HEREDA DE D
    CLASE MOTO HEREDA DE D
```

La **herencia híbrida** combina modelos de herencia.

Ejemplo en **pseudocódigo**:

```
CLASE A
    CLASE B HEREDA DE A
    CLASE C HEREDA DE A

CLASE D HEREDA DE B Y HEREDA DE C
```

Polimorfismo

El **polimorfismo** consiste en que **las clases hijas implementan la misma función pero hacen distinta cosa**.

Ejemplo en **Java**:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.diHola();
    }
}

class Vehiculo {

    public void diHola() {
        System.out.println("Hola!!");
    }
}

class Coche extends Vehiculo {
    public void diHola() {
        System.out.println("Soy un coche");
    }
}
```

Una misma función implementada en dos clases con una relación de dependencia ejecutará la función que esté más cercana a ese nivel de dependencia, es decir, si creamos una instancia de una clase y tenemos la misma función en las dos clases, ejecutará la función de la clase hija.

También existe el **polimorfismo a nivel de función**, es decir, una misma función puede aparecer múltiples veces en la misma clase devolviendo valores diferentes, aceptando parámetros diferentes o haciendo cosas diferentes.

Ejemplo de polimorfismo a nivel de función:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.sumaNumeros(3, 5);
        coche.sumaNumeros((float)5, (float)20);
        coche.sumaNumeros(2.3,1.5);
    }
}

class Vehiculo {

    public void diHola() {
        System.out.println("Hola!!");
    }
}

class Coche extends Vehiculo {
    public void diHola() {
        System.out.println("Soy un coche");
    }

    public int sumaNumeros(int a, int b) {
        System.out.println("Soy el suma números de INT");
        return a + b;
    }

    public float sumaNumeros(float a, float b) {
        System.out.println("Soy el suma números de FLOAT");
        return a + b * (float)9.0;
    }

    public void sumaNumeros(double a, double b) {
        System.out.println("Soy el suma números de DOUBLE");
        System.out.println("El resultado es: " + (a + b));
    }
}
```

Interfaces

Las **interfaces** son parecidas a las clases abstractas, pero a diferencia de estas **no implementan ninguna función (ni nos dan propiedades), sino que nos dicen lo que nosotros tenemos que implementar**. Las interfaces no se pueden usar directamente, debemos crear una clase que implemente a la interfaz.

Es una forma que yo tengo de indicar a un programador que cuando implemente una clase tiene que implementar los métodos definidos en la interfaz. En el momento en el que yo implemento las funciones definidas en la **interfaz** se considera que esta está implementada.

Ejemplo de **interfaz**:

```
public class Main {  
  
    public static void main(String[] args) {  
    }  
}  
  
// Todas las clases que implementen la interfaz Vehiculo tienen que tener estas dos funciones  
interface Vehiculo {  
  
    void Acelerar(int cuantaVelocidad);  
    void Frenar(int cuantaVelocidad);  
}  
  
// Vamos a crear una clase Coche que implementa la interfaz Vehiculo  
class Coche implements Vehiculo {  
  
    public void Acelerar(int cuantaVelocidad) {  
    }  
  
    public void Frenar(int cuantaVelocidad) {  
    }  
}
```

Las interfaces se usan para unificar métodos, estos métodos los vamos a implementar en una clase o más clases y luego los vamos a invocar. La ventaja principal de la interfaz, es que si yo se que mi clase implementa una interfaz y se que otra clase mia implementa la misma interfaz, tengo la garantía que tanto la clase 1 como la clase 2 voy a tener las mismas funciones.

10. MÉTODOS DE CLASE

Los **métodos de clase** son funciones dentro de clases. Estos métodos tienen lo que se denomina prototipo (también signature o firma) que es la forma en la que la declaramos, como el **ámbito** (**public** que puede ser utilizado desde fuera de la clase -como podría ser un objeto, una instancia de nuestra clase- o **private** que solo se puede utilizar dentro de la propia clase), el **nombre del método**, los **parámetros** (cada uno con su tipo de dato correspondiente) y **puede devolver** un valor.

Ejemplo de firma de un método:

```
CLASE A
    PROPIEDAD1
    PROPIEDAD2

    [VISIBILIDAD] [NOMBRE_DEL_METODO] ([PARAMETROS]) [VALOR]
    PUBLICA leerLibros(TEXTO libro) TEXTO contenido
```

Ejemplo en Java:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        coche.Acelerar(50);
        var resultado = suma(2, 5);
        System.out.println(resultado);
    }
    public static int suma(int operandoA, int operandoB) {
        return operandoA + operandoB;
    }
}

interface Vehiculo {

    void Acelerar(int cuantaVelocidad);
    void Frenar(int cuantaVelocidad);
}

class Coche implements Vehiculo {

    public void Acelerar(int cuantaVelocidad) {
    }
    public void Frenar(int cuantaVelocidad) {
    }
}
```

Ejemplo requiriendo como parámetro una interfaz:

```
public class Main {  
  
    public static void main(String[] args) {  
    }  
    public static void EjecutaAcelerar(Vehiculo vehiculo) {  
        // Solicitamos como parámetro una interfaz  
    }  
}  
  
interface Vehiculo {  
    void Acelerar(int cuantaVelocidad);  
    void Frenar(int cuantaVelocidad);  
}  
  
class Coche implements Vehiculo {  
    public void Acelerar(int cuantaVelocidad) {  
        System.out.println("Coche() -> Acelerar()");  
    }  
    public void Frenar(int cuantaVelocidad) {  
        System.out.println("Coche() -> Frenar()");  
    }  
}
```

Cuando estoy requiriendo una interfaz, realmente **no estoy pidiendo** al programador una interfaz, **estoy pidiendo** que me pase como parámetro el nombre de una clase que implemente esa interfaz.

Se dice que **una clase satisface a una interfaz** cuando una clase implementa todos los métodos de esa interfaz.

Hay algunos lenguajes de programación que al implementar una interfaz no se comprueba que estén todas las funciones implementadas en la clase, en el caso de **Java** si comprueba que estén todas las funciones de la interfaz en la clase.

Wrapper

Las **funciones que envuelven** se denominan **wrapper**.

Ejemplo de envoltorio o wrapper:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        Moto moto = new Moto();
        EjecutaAcelerar(moto);
        EjecutaAcelerar(coche);
    }
    public static void EjecutaAcelerar(Vehiculo vehiculo) {
        vehiculo.Acelerar(15);
    }
}

interface Vehiculo {
    void Acelerar(int cuantaVelocidad);
    void Frenar(int cuantaVelocidad);
}

class Coche implements Vehiculo {
    public void Acelerar(int cuantaVelocidad) {
        System.out.println("Coche() -> Acelerar()");
    }
    public void Frenar(int cuantaVelocidad) {
        System.out.println("Coche() -> Frenar()");
    }
}

class Moto implements Vehiculo {
    public void Acelerar(int cuantaVelocidad) {
        System.out.println("Moto() -> Acelerar()");
    }
    public void Frenar(int cuantaVelocidad) {
        System.out.println("Moto() -> Frenar()");
    }
}
```

Los **objetos (coche y moto)** van a tener las funciones **Acelerar()** y **Frenar()**, porque están implementando la interfaz **Vehiculo**, ya que hemos implementado la clase **Moto** y **Coche** a partir de la interfaz **Vehiculo** y a partir de estas clases hemos creado los **objetos (coche y moto)**.

Creamos un método **EjecutarAcelerar()** para envolver un objeto (que será el parámetro de la función) que satisfaga a la interfaz **Vehiculo**.

Le tenemos que pasar un parámetro de nombre **vehiculo** del tipo de dato **Vehiculo** a la función. Al hacer uso de la función **EjecutarAcelerar()** le pasamos como parámetro el objeto del cual vamos a utilizar su función en **vehiculo.Acelerar()**. Si cambiamos en **EjecutarAcelerar(moto)**, la línea siguiente sería **moto.Acelerar()**.

Dependiendo del parámetro que estoy recibiendo ejecuta el método **Acelerar()** en la clase **Coche** o en la clase **Moto**.

Los **parámetros que le pasamos a una función** pueden ser de dos tipos: el **paso por valor** y el **paso por referencia**.

Paso por valor

Cuando llamamos a una función copiamos los valores en memoria y se los pasamos.

El **paso por valor** lo que hace es que copian los valores al invocar al método en una zona de memoria distinta y por tanto los valores originales (`valA = 5` y `valB = 10`) no se modifican. La desventaja del **paso por valor** es que duplicamos el uso de memoria.

Ejemplo de **paso por valor**:

```
public class Main {  
  
    public static void main(String[] args) {  
        int valA = 5;  
        int valB = 10;  
  
        suma(valA, valB);  
  
        System.out.println(valA);  
        System.out.println(valB);  
    }  
  
    public static int suma(int a, int b) {  
        return a + b;  
    }  
}
```

Paso por referencia

El **paso por referencia** le pasamos la dirección de memoria de la variable y no lo copia, manipula la variable que ya tenemos almacenada.

En Java se puede usar los **pasos por referencia** de forma no tan visible como en otros lenguajes. En Java no existen los punteros, en otros lenguajes tenemos los punteros y estos pasan explícitamente cuando pasamos por valor y cuando por referencia.

En Java pasamos por referencia cuando el parámetro de una función es un objeto, porque le estamos pasando la referencia del objeto.

Ejemplo de paso por referencia:

```
public class Main {

    public static void main(String[] args) {
        Coche coche = new Coche();
        cocheChanger(coche);
        cocheChanger(coche);
        System.out.println(coche.velocidad);
    }

    public static void cocheChanger(Coche coche) { // paso por referencia
        coche.velocidad += 50; // aumento en 50 la propiedad velocidad
    }

    public static int suma(int a, int b) { // paso por valor
        return a + b;
    }
}

interface Vehiculo {
    void Acelerar(int cuantaVelocidad);
    void Frenar(int cuantaVelocidad);
}

class Coche implements Vehiculo {
    int velocidad = 0;
    public void Acelerar(int cuantaVelocidad) {
        System.out.println("Coche() -> Acelerar()");
    }
    public void Frenar(int cuantaVelocidad) {
        System.out.println("Coche() -> Frenar()");
    }
}
```

Al ser un paso por referencia, la variable original `coche` se altera.

En un **paso por valor** copiamos la variable en otra zona de memoria, **manipulamos esa zona de memoria nueva y la destruyo** después de utilizarlo, mientras que un **paso por referencia** manipula la **zona de memoria** de un objeto o una variable ya existente, no lo copiamos a otra zona de memoria nueva, lo manipulamos y se acabó, **cogemos la zona original de memoria, la manipulamos y el valor permanece cambiado**.

Recursividad

La **recursividad** consiste en que **un método se va a llamar a sí mismo** una y otra vez. El problema de las funciones recursivas es que si no controlamos bien lo que estamos haciendo va a llegar un momento en el que va a reventar el programa. La recursividad tenemos que saber cuando detenerla.

Ejemplo de recursividad **con pseudocódigo**:

```
FUNCION SUMA(INT A, INT B) {  
    VAR TEMP = A + B  
  
    SI TEMP ES MAYOR A 15  
        PARA!  
/* o también  
    SI TEMP NO ES MAYOR A 15  
        SUMA (A, TEMP)  
*/  
    SUMA(A, TEMP)  
}  
VAR VALA = 5;  
VAR VALB = 10;  
SUMA(VALA, VALB)
```

Ejemplo de recursividad **con Java**:

```
public class Main {  
  
    public static void main(String[] args) {  
        factorial(8);  
    }  
    public static int factorial(int numero) { // función recursiva  
        int resultado;  
        if (numero == 1) {  
            return 1;  
        }  
        resultado = factorial(numero -1) * numero;  
        return resultado;  
    }  
    public static int factorialNR(int numero) { // función no recursiva  
        int temp;  
        int resultado = 1;  
  
        for (temp = 1; temp <= numero; temp++) {  
            resultado = resultado * temp;  
        }  
        return resultado;  
    }  
}
```

11. LENGUAJES COMPILADOS E INTERPRETADOS

¿Cómo funciona un compilador?

Un compilador lo que hace es coger una secuencia de caracteres que forman un fichero de código y lo convierten en código máquina o puede ser otro lenguaje.

Hay dos tipos de compiladores, los que generan código máquina o los que generan código intermedio (o representación intermedia).

El lenguaje compilado se compila a código máquina y un lenguaje interpretado no.

Cómo trabaja el compilador:

CÓDIGO FUENTE → [AQUÍ TRABAJA EL COMPILADOR] → CÓDIGO FINAL

La **primera parte** de un compilador es lo que se llama un **analizador léxico**. Su función principal es generar una cosa que se llama **string de tokens** (o tokens). En la siguiente línea, el analizador léxico irá leyendo cada una de las letras y lo tokeniza.

```
var variable = 10;
v → letra v
a → letra a
= → símbolo igual
1 → número
0 → número
; → punto y coma
```

Luego el **analizador léxico** sigue analizando:

```
var → "palabra reservada var" (Ej.: if, while, do, static,...)
variable → "identificador" (las variables se llaman identificadores también)
= → "símbolo_asignación"
10 → "número_entero"
```

La **siguiente fase** de un compilador es la fase de **análisis sintáctico**. Aquí va leyendo los tokens creados por el analizador léxico y analiza si tiene sentido lo que queremos poner.

```
if (numero > 10) { ... }
1. if
2. abre_parentesis
3. condición
4. cierra_parentesis
```

5. abre llave
6. expresiones
7. cierra llave

Una vez que ha finalizado el **análisis sintáctico** y todo es correcto, se genera un código intermedio (antes del código final) que simplifica el lenguaje humano a un lenguaje mejor representado en estructuras de datos y que el compilador lo puede manipular mejor.

En esta fase de código intermedio se genera una estructura de datos que se llaman **abstract syntax tree** (árbol sintáctico abstracto). Este es una estructura en forma de árbol que representa nuestro código, pero a nivel de estructura de datos.

```
5 + 2 + 4
   +
   5          // Se lee de abajo a arriba y de izquierda a derecha
 +
2   4
```

Ahora **pasamos a la fase** de **optimización del código intermedio**. El código intermedio es una traducción literal del lenguaje humano a un lenguaje que entiende el compilador, pero los compiladores optimizadores (que son prácticamente todos) pueden ser capaces de generar un código más óptimo que el que nosotros hemos escrito o de eliminar código redundante.

Si tuviéramos el siguiente código:

```
var variable = 3;
if (3 == 0) {
    accion
}
```

Un compilador optimizador sería capaz de eliminarlo del código intermedio (y no pasaría al código final), ya que como la condición nunca se cumpliría y no se ejecuta la acción.

También se puede dar el caso de que tenemos un bucle:

```
for (i = 0; i <= 4; i++) {
    System.out.println("Valor actual: " + i)
}
```

El compilador puede sustituir la iteración (por ser más costosa) por el código siguiente:

```
System.out.println("Valor actual: 0")
System.out.println("Valor actual: 1")
System.out.println("Valor actual: 2")
System.out.println("Valor actual: 3")
```

Esto se denomina en un compilador optimizador **unroll loops** (desenrollar el for).

Ahora pasamos a la fase de **generación de código**, que es donde vemos los resultados. Aquí genera un código máquina o un código para ser interpretado.

También **podríamos tener una fase de optimización del código generado**, que mediante una serie de algoritmos podría optimizar su propio código. Esto no todos lo hacen, pero si la mayoría.

¿Compilado o interpretado?

Diferencia entre un lenguaje compilado a código máquina y otro interpretado (compilado para ser interpretado). El **compilado** se compila a código máquina y se ejecuta directamente en nuestro procesador a través del sistema operativo. El **interpretado** no son ejecutados directamente por el sistema operativo o nuestro procesador, se ejecutan a través de una máquina virtual o intérprete.

Java es un lenguaje interpretado a través del JVM (Java Virtual Machine), porque no se ejecuta de forma nativa en el procesador. El compilador genera un código que se llama Bytecode que es el que luego se ejecuta en la JVM.

Bytecode → JVM → Resultado final

Ejemplos de **lenguajes compilados**: C, C++, Rust, Go, ensamblador,...

Ejemplos de **lenguajes interpretados**: Java, Python, PHP, Perl, TCL, Dart, Ruby,...

Tenemos también otros lenguajes que son **compilados e interpretados a la vez**. En el caso de C# se genera un código llamado MSIL (Microsoft Intermediate Language) que se interpreta y las partes de código que se ejecuta muchas veces lo compila para hacerlo más eficiente. Esta técnica se llama **JIT** (Just In Time) o **JIT Compiler**.

Los **lenguajes compilados son mucho más rápidos** que los lenguajes interpretados, pero estos son más fáciles de depurar.

Los lenguajes **interpretados se pueden ejecutar en cualquier máquina**, los compilados solo se pueden ejecutar en la máquina para la que han sido compilados.