

CFGS Desarrollo de aplicaciones multiplataforma

Módulo profesional: Programación



**GENERALITAT
VALENCIANA**

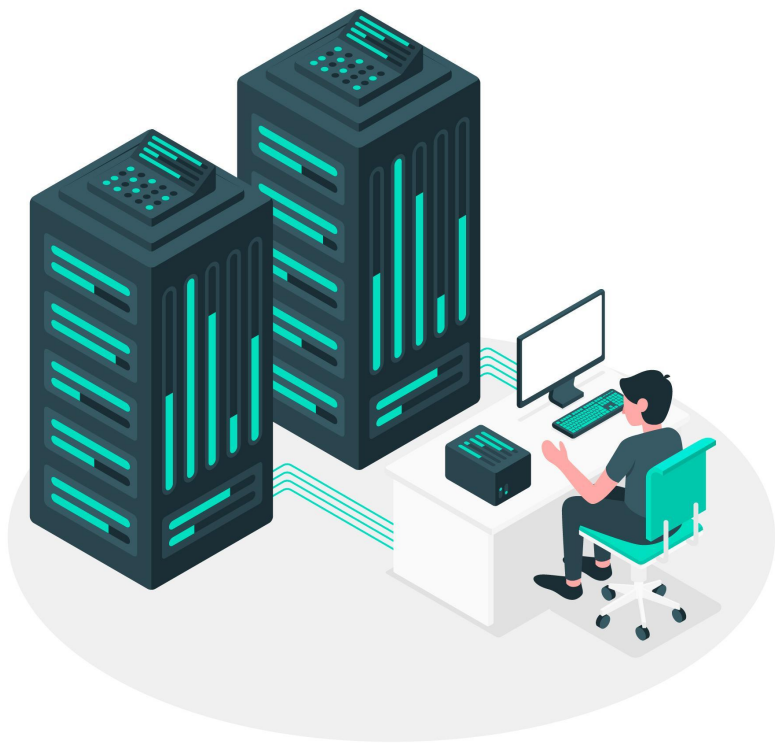
Conselleria d'Educació,
Investigació, Cultura i Esport



Unió Europea

Fons Social Europeu

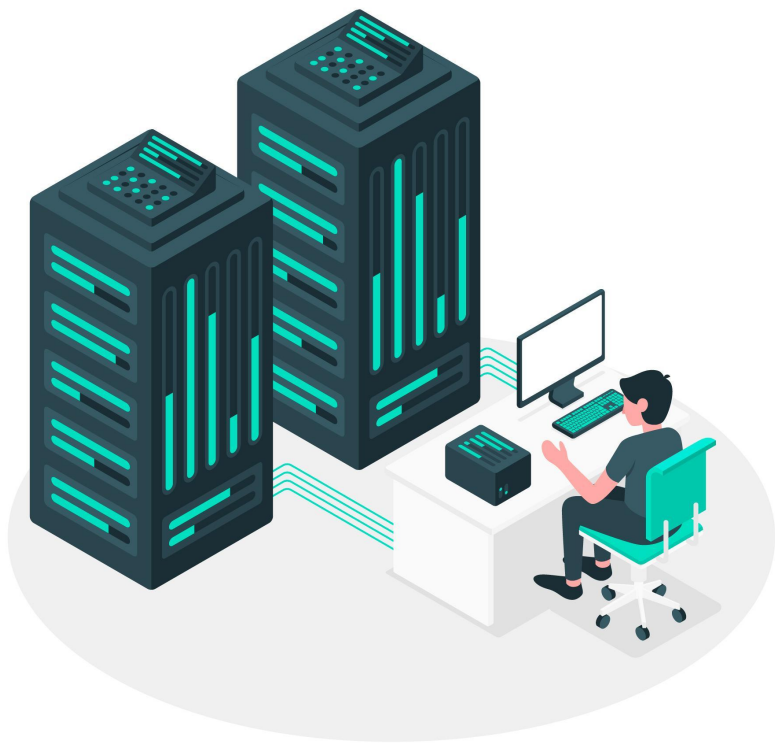
L'FSE inverteix en el teu futur



Datos profesor

Joan Carrillo

Web del módulo: <https://aules.edu.gva.es/semipresencial/login/index.php>



Material elaborado por:

Edu Torregrosa Llácer

modificado:

Joan Carrillo

Esta obra está licenciada bajo la licencia **Creative Commons Atribución-NoComercial-CompartirIgual 4.0 internacional**. Para ver una copia de esta licencia visita: <https://creativecommons.org/licenses/by-nc-sa/4.0/>



**Attribution-NonCommercial-ShareAlike
4.0 International (CC BY-NC-SA 4.0)**

Introducción a JAVA



1. Introducción a JAVA.
2. Preparación del entorno: JDK y VS Code.
3. Mi primer algoritmo en JAVA.
4. Variables.
5. Cadenas de texto.
6. Operadores y expresiones.
7. Condicionales.
8. Bucles.
9. Entrada de datos.
10. Conversión de tipos.

El origen de JAVA

En 1985, Sun Microsystems estaba intentando desarrollar una nueva tecnología para programar todo tipo dispositivos inteligentes de nueva generación. El equipo en principio considero utilizar C++, pero decidieron rechazarlo por varios motivos:

- Para sistemas con recursos limitados, C++ necesitaba demasiada memoria.
- Excesiva complejidad, en C++ los programadores tenían que administrar manualmente la memoria del sistema, una tarea compleja y propensa a errores. Para liberar de esta tarea al programador JAVA desarrolló un **Garbage collector**, que permitía una asignación y liberación automática de la memoria a medida que se ejecutaba un programa.
- Querían una plataforma que pudiera adaptarse fácilmente a todo tipo de dispositivos.

JAVA realmente nace de la mano de James Gosling en 1989 bajo el nombre de “Oak”, aunque más tarde por una serie de problemas legales, termina con el nombre de JAVA.

El lenguaje JAVA

En definitiva, el objetivo de JAVA era crear un lenguaje similar a C++ que facilitara la tarea al programador, con orientación a objetos y con una máquina virtual propia. Se pretendía desarrollar un lenguaje que pudiera ser usado sobre cualquier arquitectura y sobre cualquier tipo de dispositivo. Es por ello que se crea bajo el principio “*Write Once, Run Anywhre*” (escribelo una vez, ejecútalo en cualquier sitio).

Tras unos años en desarrollo, Sun Microsystems finalmente lo presentó en 1995, y poco después, en 1996, se lanza la primera versión, el JDK 1.0. Actualmente JAVA es uno de los lenguajes más utilizados y está instalado en miles de millones de dispositivos.

A partir de ese momento han ido apareciendo nuevas versiones con continuas mejoras sobre el lenguaje. La última versión disponible es **JAVA SE 19**, lanzada en septiembre de 2022.

El lenguaje JAVA

Algunas de las principales características de JAVA son las siguientes:

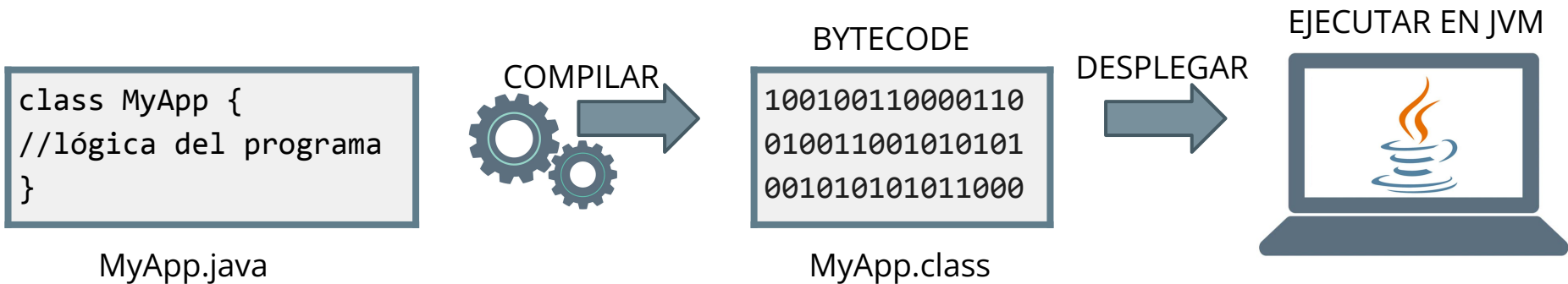
- Es un lenguaje **orientado a objetos**.
- En JAVA cada programa únicamente se escribe una vez y se compila una vez. La ejecución de los programas JAVA es **independiente** de la plataforma. *“write once, run anywhere”*.
- El secreto de su gran versatilidad reside en la Máquina Virtual JAVA (JVM). La **JVM** es una extensión del sistema real con el que se trabaja, que permite ejecutar el código resultante de un programa JAVA ya compilado, independientemente de la plataforma donde se ejecute.
- Es necesario disponer de una JVM en la plataforma desde donde vayamos a ejecutar programas diseñados en JAVA.

¿Cómo funciona JAVA?

El proceso es el siguiente:

En primer lugar creamos un archivo con extensión **java**, que es el código fuente de nuestro algoritmo. Más tarde ese código es compilado y obtenemos un archivo con extensión **class**. Ese archivo class es el **BYTECODE JAVA**.

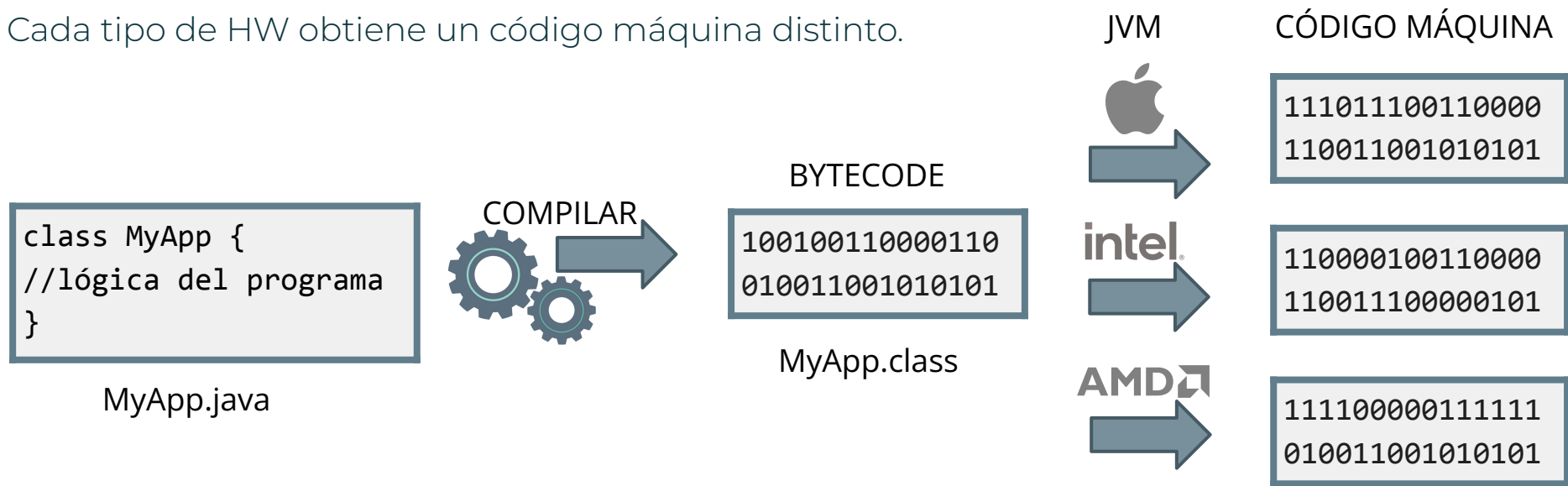
A continuación, ese BYTECODE JAVA es interpretado por la Máquina virtual JAVA, y mediante un compilador JIT se obtiene el **código máquina**. En último lugar, el código máquina es ejecutado por el Hardware del dispositivo.



¿Cómo funciona JAVA?

Ese archivo class en **BYTECODE JAVA** es exactamente el mismo, independientemente de la plataforma. Es la JVM la que se encarga de interpretar ese BYTECODE y convertirlo en código máquina específico para ese hardware (dispositivo/arquitectura).

Cada tipo de HW obtiene un código máquina distinto.



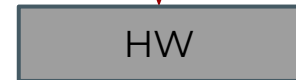
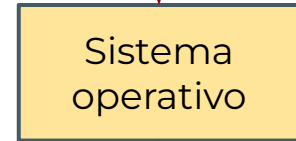
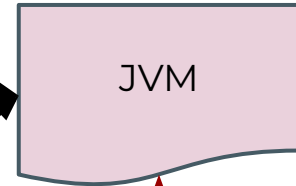
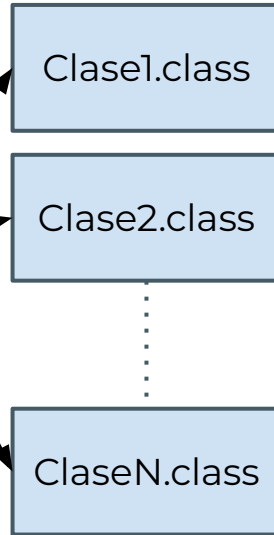
El lenguaje JAVA

Compilación y ejecución en JAVA

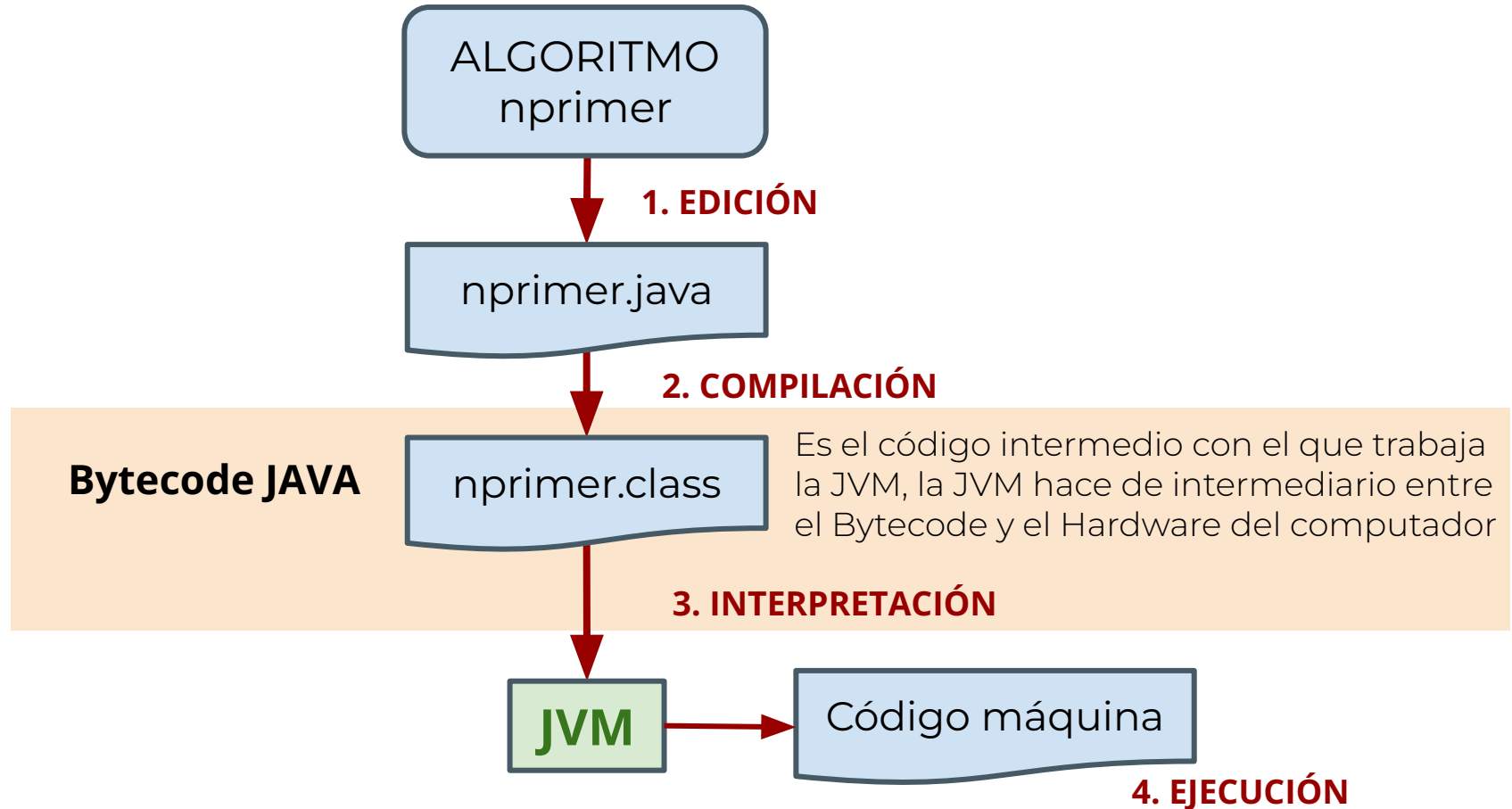
CÓDIGO FUENTE



CÓDIGO OBJETO



El lenguaje JAVA



Preparación del entorno: Requisitos

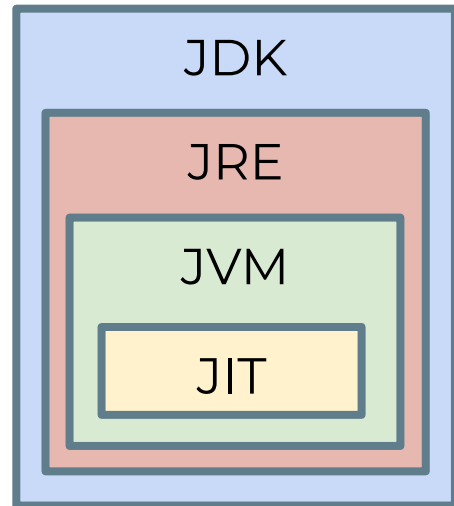
Programas en JAVA

- ¿Qué hace falta para usar un programa creado en Java?

- Java Runtime Environment (**JRE**)
- Máquina virtual JAVA (**JVM**)

- ¿Qué nos hace falta para crear un programa en Java?

- Java Development Kit (**JDK**)
- Editor de código, en nuestro caso **VSCode**



[Información adicional sobre el compilador JIT](#)

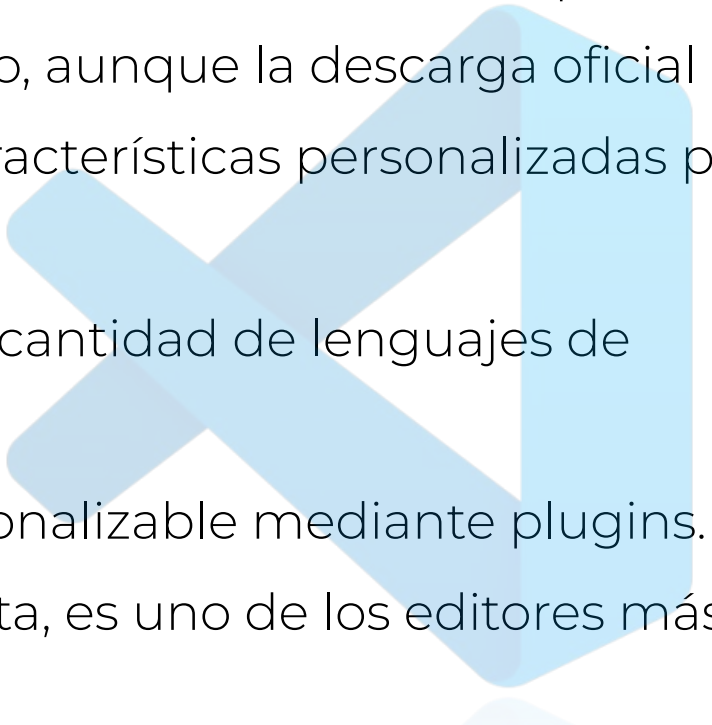
Java Development Kit (JDK)

- El JDK (Java Development Kit) es la herramienta básica para crear programas usando el lenguaje Java. Se puede descargar desde la página oficial de Java, en el sitio web de Oracle:

<https://www.oracle.com/java/technologies/downloads/>

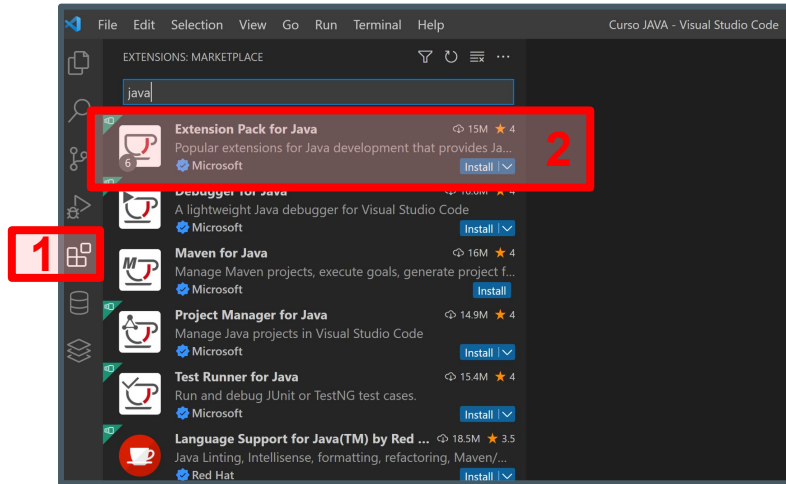
Java 19 <u>Java 17</u>		
Java SE Development Kit 17.0.4.1 downloads		
Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications and components using the Java programming language.		
The JDK includes tools for developing and testing programs written in the Java programming language and running on the Java platform.		
Linux <u>macOS</u> Windows		
Product/file description	File size	Download
Arm 64 Compressed Archive	171.63 MB	https://download.oracle.com/java/17/latest/jdk-17_linux-aarch64_bin.tar.gz (sha256)
Arm 64 RPM Package	153.65 MB	https://download.oracle.com/java/17/latest/jdk-17_linux-aarch64_bin.rpm (sha256)
x64 Compressed Archive	172.85 MB	https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.tar.gz (sha256)
x64 Debian Package	148.43 MB	https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.deb (sha256)
x64 RPM Package	155.26 MB	https://download.oracle.com/java/17/latest/jdk-17_linux-x64_bin.rpm (sha256)

Visual Studio Code

- Visual Studio Code es un editor de código fuente desarrollado por Microsoft. Es gratuito y de código abierto, aunque la descarga oficial está bajo software privativo e incluye características personalizadas por Microsoft.
 - Este editor es compatible con una gran cantidad de lenguajes de programación, entre ellos, JAVA.
 - Es un editor modular y totalmente personalizable mediante plugins.
 - Pese a tener una vida relativamente corta, es uno de los editores más utilizados en la actualidad.
- 

Visual Studio Code

- Accederemos a su web para iniciar la **descarga**:
<https://code.visualstudio.com/>
- Tras ello, procederemos con su instalación. Una vez instalado, hacemos clic en extensiones y buscamos e instalamos la extensión de Java.



Primeros pasos en JAVA

Mi primer programa en JAVA - Hola mundo!!

Quien ya conozca otros lenguajes de programación, verá que hacer un **Hola mundo !!** en Java parece más complicado.

Por ejemplo, en BASIC bastaría con escribir **PRINT "Hola mundo!"**. Pero esta mayor complejidad inicial es debida al "cambio de mentalidad" que tendremos que hacer cuando empleamos JAVA, y dará lugar a otras ventajas más adelante, cuando nuestros programas sean mucho más complejos.

```
//  
// Aplicación HolaMundo de ejemplo  
//  
class HolaMundo {  
    public static void main( String args[] ) {  
        System.out.println( "Hola Mundo!" );  
    }  
}
```

Mi primer programa en VS Code

1. En primer lugar, deberemos crear una carpeta donde guardaremos el código Java. Una buena estructura sería crear una carpeta **CursoJava** y dentro de ella una carpeta por cada tema, en este caso crearemos la carpeta **IntroduccionAJava**.
2. Tras ello abriremos la carpeta con la opción Open folder...
3. Ahora creamos un nuevo fichero con la opción Archivo| Nuevo archivo y procederemos a guardarlo con la opción Archivo| Guardar como... le diremos HolaMundo.java
4. En el fichero que hemos creado copiaremos el código Java que tenemos a continuación.

```
//  
// Aplicación HolaMundo de ejemplo  
//  
class HolaMundo {  
    public static void main( String args[] ) {  
        System.out.println( "Hola Mundo!" );  
    }  
}
```

Mi primer programa en VS Code

5. Es el momento de compilar el programa, lo haremos en la terminal que tiene Code, opción

Terminal > Nuevo Terminal

6. Procedemos a compilar con la siguiente orden:

```
javac HolaMundo.java
```

7. La instrucción anterior nos habrá creado el BYTECODE de JAVA, que es el archivo **HolaMundo.class**. Finalmente para ejecutar el programa debemos introducir:

```
java HolaMundo
```

8. Lo cual nos mostrará por consola el mensaje Hola Mundo!

```
Hola Mundo!
```

Entendiendo mi primer programa en JAVA

Comentario, sirven para mejorar la lectura del código, no se ejecutan

Declaración de la clase 'HolaMundo'

Declaración del método 'main'

```
//  
// Aplicación HolaMundo de ejemplo  
//  
class HolaMundo {  
    public static void main( String args[] ) {  
        System.out.println( "Hola Mundo!" );  
    }  
}
```

Instrucción que imprime por pantalla una cadena de texto, las instrucciones se separan con punto y coma

Fin del método 'main'

Fin de la clase 'HolaMundo'

Salida por pantalla

Salida por pantalla

La salida se realiza a partir de la clase **System.out**. A partir de dicha clase podemos acceder a los siguientes métodos:

print: Imprime por pantalla una cadena de tipo String. Se pueden utilizar otros tipos básicos de variables, ya que se hace una conversión a partir de su valor. Además podemos utilizar el operador “+” para concatenar Strings. Todos los literales deben ir entre comillas dobles, a excepción de un único carácter, que en ese caso puede ir también entre comillas simples.

```
System.out.print("Hola");
```

println: Igual que print, pero añade un salto de línea.

```
System.out.println("Hola con salto de línea");
```

Salida por pantalla con variables

Podemos **concatenar** texto entre comillas con los valores de las variables, para ello debemos utilizar el **operador suma (+)**. Por ejemplo:

```
int resultado = 20;  
System.out.print("El resultado es"+resultado);
```

- La salida del programa será: → **El resultado es20**
- Se deben tener en cuenta los espacios en blanco para separar el texto de los valores de las variables. Además, también podemos concatenar texto con el resultado de una expresión. Por ejemplo:

```
System.out.println("El resultado es "+resultado+5);           //205  
System.out.println("El resultado es "+(resultado+5));         //25  
System.out.println(resultado+5);                               //25
```


Variables en JAVA

Variables

- Las **variables** se utilizan para almacenar los datos que manejan los programas. Pueden guardar todo tipo de información, desde cantidades numéricas y textos, hasta valores booleanos.
- Como el propio nombre indica, los datos almacenados en dichas variables pueden “variar” a lo largo de la ejecución del programa.
- Necesitamos un **espacio de memoria** donde poder almacenar los valores que tienen almacenados las variables en cada momento.
- En casi cualquier lenguaje de programación podremos reservar esos "espacios", y asignarles un nombre con el que acceder a ellos. Esto es lo que se conoce como "variables".
- Para no desperdiciar la memoria de nuestro ordenador, el espacio de memoria que hace falta "reservar" será distinto según el “tipo de datos” que queramos almacenar(enteros, reales, caracteres, cadenas, etc).

Tipado fuerte

JAVA es un lenguaje con un tipado fuerte, esto significa que se debe indicar el tipo de datos que queremos guardar dentro de una variable. El tipo de tipado es una característica propia de cada lenguaje.

Esto hace a JAVA más “estricto” en la declaración de variables, esto normalmente se traduce en una mayor protección contra posibles errores.

Por ejemplo:

- Declarar una variable que se llama **cantidad** y es de **tipo entero**
- Declarar una variable que se llama **nombre** y es de **tipo texto**

En otros lenguajes como JavaScript simplemente indicaremos que vamos a declarar dos variables, que se llaman nombre y cantidad.

Almacenamiento de la información

Antes de empezar con los detalles de las variables en JAVA, es necesario saber **cómo se almacena la información en la memoria de los dispositivos**.

En computación toda la información se almacena en **código binario**, es decir, en base a dos valores, el cero y el uno. A esta unidad mínima de información se le denomina **bit**. Es decir, en un bit, únicamente podemos almacenar un cero o un uno. Normalmente cuando hablamos de capacidad de almacenamiento, lo expresamos en bytes, **1 byte son 8 bits**.

Dependiendo de la cantidad de bits que utilicemos, tendremos más combinaciones distintas, esto nos proporciona la capacidad de almacenar todo tipo de valores. **Por ejemplo:**

En 2 bits tenemos 4 combinaciones.



0	0	Podemos almacenar un 0
0	1	Podemos almacenar un 1
1	0	Podemos almacenar un 2
1	1	Podemos almacenar un 3

Almacenamiento de la información

En **2 bits** tenemos
4 combinaciones:

0	0
0	1
1	0
1	1

En **3 bits** tenemos
8 combinaciones:

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

En **4 bits** tenemos **16**
combinaciones:

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

**n bits = 2^n
combinaciones**

**1 byte = 2^8
combinaciones**

Tipos de variables numéricas en JAVA

Los tipos de variables numéricas disponibles en JAVA son los siguientes:

Nombre	¿Admite decimales?	Valor mín		Valor máx		Precisión	Ocupa
byte	no	-128	-2^7	127	2^7-1	-	1 byte
short	no	-32768	-2^{15}	32767	$2^{15}-1$	-	2 bytes
int	no	-2.147.483.648	-2^{31}	2.147.483.647	$2^{31}-1$	-	4 bytes
long	no	-9.223.372.036 .854.775.808	-2^{63}	9.223.372.036 .854.775.807	$2^{63}-1$	-	8 bytes
float	si	-3.4E38		3.4E38		7 cifras	4 bytes
double	si	-1,7E308		1,7E308		16 cifras	8 bytes

Existen tipos adicionales como **BigInteger** o **BigDecimal**, que se utilizan para almacenar números muy grandes o para una mayor precisión. Aunque **no son tipos primitivos** en JAVA.

Almacenamiento de la información

Ejemplo de representación interna del **número 3** en cada uno de los tipos de datos enteros:

Tipo de dato: **byte**

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

 8 bits

Tipo de dato: **short**

[illegible]

Tipo de dato: **int**

[illegible]

Tipo de dato: **long**

[illegible]

Otros tipos de variables en JAVA

Tenemos otros tipos básicos de variables, que no son para datos numéricos:

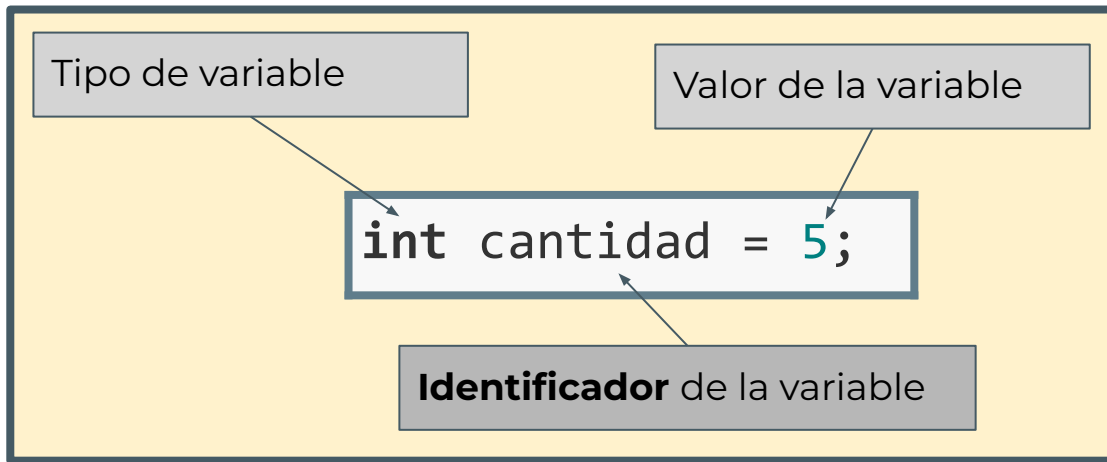
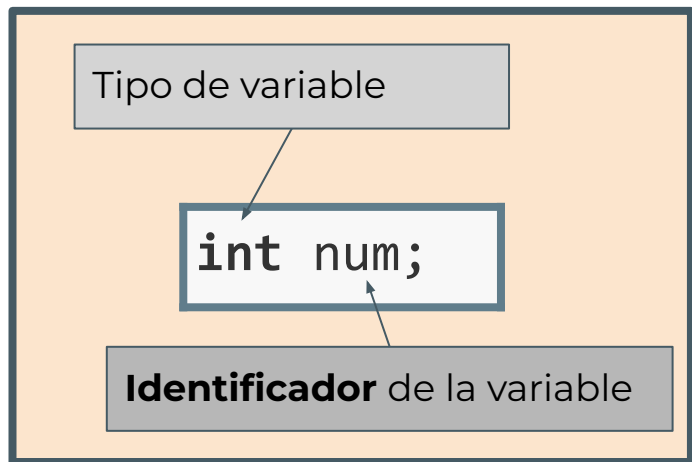
- **char**, almacena un único carácter, puede ser una letra, un dígito numérico, o cualquier otro carácter alfanumérico. Ocupa 2 bytes y sigue un estándar llamado Unicode (que a su vez engloba a otro estándar anterior llamado ASCII).
- **boolean**, se usa para evaluar condiciones, y puede tener el valor "verdadero" (true) o "falso" (false). Ocupa 1 bit.
- **String**, se utiliza para almacenar cadenas de caracteres, es decir, puede almacenar un conjunto de char. Aunque **no es un tipo primitivo**, es un tipo de dato muy utilizado.

Declarar variables en JAVA

Declarar variables en JAVA

La forma de "declarar" una variable en JAVA es detallando primero el tipo de datos que podrá almacenar y después el nombre(identificador) que daremos la variable. Además, se puede indicar un valor inicial a través del igual.

Para finalizar la asignación de la variable debemos introducir un punto y coma.



Declarar variables en JAVA

La declaración de variables funciona de la siguiente forma:

1. Se evalúa la expresión a la derecha del igual.
2. Se guarda el resultado de la expresión sobre la variable a la izquierda del igual.

Además podemos asignar a una variable el valor de otra variable, e incluso, realizar operaciones con ellas.

```
int cantidad = 5 + 2; //guardamos en cantidad un 7
int precio = 10; //guardamos en precio un 10
precio = 5; //modificamos el precio a 5
int precioFinal = cantidad * precio; //guardamos un 35 en precioFinal
precioFinal = precioFinal + 10; //guardamos 45 en precioFinal
```

Declarar varias variables en JAVA

Se pueden declarar varias variables del mismo tipo "a la vez", separando con comas:

```
int a = 5, b = 10; // "a" y "b" son enteros y valen 5 y 10
short c = -1, d, e = 4; // "c" vale -1, "d" no tiene valor, "e" vale 4
int f, g; // declaramos las variables "f" y "g" de tipo entero
```

Los **nombres de variables** pueden contener letras y números (pero no pueden comenzar con un número). Pueden contener símbolos, como el de subrayado, pero no podrán contener otros muchos símbolos, como los de las distintas operaciones matemáticas posibles (+,-,*,/), ni llaves o paréntesis.

```
int a2 = 5; //correcto
int 2a = 10; //error
int nombre-2 = 20 //error
```

Declarar variables en JAVA

Identificador de la variable: es el nombre que damos a la variable. Deben seguir las siguientes reglas:

- Debe comenzar por letra, subrayado (_) o el carácter \$.
- **No** pueden **comenzar** por un **número**, pero después si podemos incluirlos.
- Las **variables son case-sensitive**, es decir diferencian entre mayúsculas y minúsculas.
- **No** pueden utilizarse **espacios en blanco ni símbolos de operadores**.
- **No** pueden coincidir con ninguna **palabra reservada**.
- El **% no** está permitido, **si el \$ y la ç**. También los acentos y la ñ, pero **no se recomienda** utilizarlos.

Palabras reservadas en JAVA

Existen palabras reservadas que **no se pueden utilizar como identificadores para las variables**:

abstract	assert	boolean	break	byte	case	catch	char
class	const	continue	default	do	double	else	enum
extends	false	final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long	native	new
null	package	private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw	throws	transient
true	try	void	volatile	while			

Declarar variables numéricas en JAVA

Para utilizar **decimales se usa el punto**, y una **'f' final** para indicar el tipo **float**. Podemos utilizar una **'d' final** para indicar que es de tipo **double**, pero no es obligatorio.

```
float f1 = 1f;  
float f2 = 5.4f;  
float f3 = 12E6f;  
float f4 = 0.55E-2f;  
float f5 = -5.44E-2f;
```

```
double d1 = 2;  
double d2 = 4.001d;  
double d3 = 1.51E-3;  
double d4 = 2.11E8d;  
double d5 = -0.11E-3;
```

Declarar variables numéricas en JAVA

Para utilizar decimales se usa el punto, y una **'f' final** para indicar el tipo **float**. Podemos utilizar una **'d' final** para indicar que es de tipo **double**, pero no es obligatorio.

```
float f1 = 1f; // "f1" es de tipo float y vale 1.0
float f2 = 5.4f; // "f2" es de tipo float y vale 5.4
float f3 = 12E6f; // "f3" es de tipo float y vale 12000000
float f4 = 0.55E-2f; // "f4" es de tipo float y vale 0.0055
float f5 = -5.44E-2f; // "f5" es de tipo float y vale -0.0544
```

```
double d1 = 2; // "d1" es de tipo double y vale 2.0
double d2 = 4.001d; // "d2" es de tipo double y vale 4.001
double d3 = 1.51E-3; // "d3" es de tipo double y vale 0.00151
double d4 = 2.11E8d; // "d4" es de tipo double y vale 211000000
double d5 = -0.11E-3; // "d5" es de tipo double y vale -0.00011
```


Declarar variables numéricas en JAVA

Podemos utilizar el guión bajo para mejorar la legibilidad del código.

```
int mil = 1_000;      // almacena 1000
int millon = 1_000_000; // almacena 1000000
```

De hecho, el guión bajo no afecta en nada a la cantidad.

```
int mil = 1_00_0;      // almacena 1000
int mil = 10_00;       // almacena 1000
int millon = 1_0_0_0_0_0_0; // almacena 1000000
int millon = 1_00_00_00;   // almacena 1000000
float mil = 1_00_0.0_1_2f; // almacena 1000.012
float millon = 1_00_00_00.0_0_5f; // almacena 1000000.005
double mil = 1_000.00_3;   // almacena 1000.003
double mil = 1_000._00_3;  // ERROR
```

El sistema hexadecimal en JAVA

Declarar variables en hexadecimal

El sistema hexadecimal es muy utilizado en computación, es un sistema de base 16. Esto significa que tiene 16 valores distintos para representar la información. **Del 0 al 9 y las letras A, B, C, D, E y F.**

Este uso tan extendido se debe a que las CPU utilizan el byte(8 bits) como unidad básica de memoria. Un byte representa 256 valores posibles, por lo tanto dos dígitos hexadecimales corresponden exactamente a un byte, $16 \times 16 = 256$. Debido a ello, podemos hacer las siguientes conversiones.

- **byte(1byte)** se representa con 2 dígitos hexadecimales (desde 0x00 hasta 0xFF)
- **short(2bytes)** se representa con 4 dígitos hexadecimales (desde 0x0000 hasta 0xFFFF)
- **int(4bytes)** se representa con 8 dígitos hexadecimales (desde 0x00000000 hasta 0xFFFFFFFF)
- **long(8bytes)** se representa con 16 dígitos hexadecimales.

Para declarar variables de tipo entero en hexadecimal debemos escribir en primer lugar **0x**:

```
byte num1 = 0x02; // almacena un 2
byte num2 = 0x0B; // almacena un 11
byte num3 = 0x12; // almacena un 18
```

Declarar variables en hexadecimal

Al tener base 16, necesitamos **4 bits** para poder representar un valor en hexadecimal:

$$2^4 = 16$$

Debemos tener en cuenta que por cada dígito en hexadecimal tenemos 16 posibles valores, por lo tanto **0x10**, es la combinación número 17, que representa al número 16.

```
byte num1 = 0x02;    // almacena un 2
byte num2 = 0x0B;    // almacena un 11
byte num3 = 0x10;    // almacena un 16
byte num4 = 0x12;    // almacena un 18
byte num5 = 0xFF;    // almacena un -1
byte num6 = 0x7F;    // almacena un 127
byte num7 = 0x80;    // almacena un -128
byte num8 = 0x012;   // almacena un 18
byte num9 = 0x112;   // ERROR
```

Podemos utilizar el hexadecimal con el resto de tipos enteros (short, int y long), pero para simplificar se han mostrado los ejemplos con el tipo byte.

0	0	0	0	= 0
0	0	0	1	= 1
0	0	1	0	= 2
0	0	1	1	= 3
0	1	0	0	= 4
0	1	0	1	= 5
0	1	1	0	= 6
0	1	1	1	= 7
1	0	0	0	= 8
1	0	0	1	= 9
1	0	1	0	= A
1	0	1	1	= B
1	1	0	0	= C
1	1	0	1	= D
1	1	1	0	= E
1	1	1	1	= F

Complemento a 2

```
byte num1 = 0x01;    // almacena un 1
byte num2 = 0x03;    // almacena un 3
byte num3 = 0x10;    // almacena un 16
```

El motivo de almacenar los enteros de esta forma (primero los positivos y luego empezar por el menor de los negativos), es que JAVA utiliza el **complemento a 2** para almacenar los enteros. Para 'n' bits el complemento a 2 proporciona el siguiente rango de valores:

Desde -2^{n-1} hasta $2^{n-1}-1$

Esto tiene varias implicaciones:

- Si el primer bit es 1 (el de más a la izquierda), se trata de un número negativo.
- La conversión de binarios positivos a negativos resulta muy sencilla, se cambian ceros por unos y finalmente se suma uno en binario
- Se facilita muchísimo la resta de números binarios al procesador (en particular a la ALU), ya que no le hace falta saber restar, simplemente suma negativos.

0	0	0	0	0	0	0	1	= 0x01 → 1
0	0	0	0	0	0	1	1	= 0x03 → 3
0	0	0	1	0	0	0	0	= 0x10 → 16

1	1	1	1	1	1	1	1	= 0xFF → -1
1	1	1	1	1	1	0	1	= 0xFD → -3
1	1	1	1	0	0	0	0	= 0xF0 → -16

0	0	0	0	= 0
0	0	0	1	= 1
0	0	1	0	= 2
0	0	1	1	= 3
0	1	0	0	= 4
0	1	0	1	= 5
0	1	1	0	= 6
0	1	1	1	= 7
1	0	0	0	= 8
1	0	0	1	= 9
1	0	1	0	= A
1	0	1	1	= B
1	1	0	0	= C
1	1	0	1	= D
1	1	1	0	= E
1	1	1	1	= F

Variables boolean y char

Declarar variables booleanas en JAVA

Las variables **booleanas** son muy utilizadas en programación y sirven para **evaluar condiciones**. Este tipo de variables únicamente pueden almacenar dos valores, **verdadero o falso**. Son muy baratas, **ocupan 1bit**.

Para declarar variables de tipo booleano se utiliza la palabra **boolean**. Para asignar valores debemos utilizar las palabras reservadas **true**, y **false**. **Las palabras reservadas nunca van entre comillas.**

```
boolean seguir = true; // correcto  
boolean parar = false; // correcto
```

Podemos asignar también el resultado de una expresión lógica a este tipo de variables:

```
boolean seguir = 3 > 5; // false  
boolean parar = 7 != 6; // true
```

Declarar variables char en JAVA

Para declarar variables de tipo char debemos poner su valor entre comillas simples:

```
char letra = 'A'; // "letra" es un carácter y tiene guardada la letra A
char simbolo = '*'; // "simbolo" es un carácter y tiene guardado el
carácter asterisco *
```

En JAVA las variables de tipo **char internamente funcionan como enteros**, de hecho podemos cambiar su valor a través de operaciones aritméticas. Los valores más frecuentes están definidos en la **tabla ASCII**.

```
char letra = 'A'; // "letra" y tiene guardada la letra A
letra = letra + 1; // "letra" y tiene guardada la letra B
letra = letra + 3; // "letra" y tiene guardada la letra E
letra = 97; // "letra" y tiene guardada la letra a
letra = 100; // "letra" y tiene guardada la letra d
```


Tabla ASCII

ASCII es un conjunto de caracteres de **7 bits** que permite representar 128 caracteres, **de 0 a 127**. ASCII representa un valor numérico para cada carácter. Por ejemplo: **A → 65**

En JAVA, ASCII es un pequeño subconjunto de los valores posibles que puede tener una variable de tipo char, Esto se debe a que JAVA utiliza **Unicode** para codificar caracteres.

Unicode se diseñó para facilitar el tratamiento informático y visualización de textos en todo tipo de idiomas. En JAVA utiliza 2 bytes, esto le permite representar 2^{16} caracteres distintos.

ASCII Hex Símbolo	ASCII Hex Símbolo	ASCII Hex Símbolo	ASCII Hex Símbolo
0 0 NUL	16 10 DLE	32 20 (space)	48 30 0
1 1 SOH	17 11 DC1	33 21 !	49 31 1
2 2 STX	18 12 DC2	34 22 "	50 32 2
3 3 ETX	19 13 DC3	35 23 #	51 33 3
4 4 EOT	20 14 DC4	36 24 \$	52 34 4
5 5 ENQ	21 15 NAK	37 25 %	53 35 5
6 6 ACK	22 16 SYN	38 26 &	54 36 6
7 7 BEL	23 17 ETB	39 27 '	55 37 7
8 8 BS	24 18 CAN	40 28 (56 38 8
9 9 TAB	25 19 EM	41 29)	57 39 9
10 A LF	26 1A SUB	42 2A *	58 3A :
11 B VT	27 1B ESC	43 2B +	59 3B ;
12 C FF	28 1C FS	44 2C ,	60 3C <
13 D CR	29 1D GS	45 2D -	61 3D =
14 E SO	30 1E RS	46 2E .	62 3E >
15 F SI	31 1F US	47 2F /	63 3F ?

ASCII Hex Símbolo	ASCII Hex Símbolo	ASCII Hex Símbolo	ASCII Hex Símbolo
64 40 @	80 50 P	96 60 `	112 70 p
65 41 A	81 51 Q	97 61 a	113 71 q
66 42 B	82 52 R	98 62 b	114 72 r
67 43 C	83 53 S	99 63 c	115 73 s
68 44 D	84 54 T	100 64 d	116 74 t
69 45 E	85 55 U	101 65 e	117 75 u
70 46 F	86 56 V	102 66 f	118 76 v
71 47 G	87 57 W	103 67 g	119 77 w
72 48 H	88 58 X	104 68 h	120 78 x
73 49 I	89 59 Y	105 69 i	121 79 y
74 4A J	90 5A Z	106 6A j	122 7A z
75 4B K	91 5B [107 6B k	123 7B {
76 4C L	92 5C \	108 6C l	124 7C
77 4D M	93 5D]	109 6D m	125 7D }
78 4E N	94 5E ^	110 6E n	126 7E ~
79 4F O	95 5F _	111 6F o	127 7F

Unicode

Tal y como hemos comentado antes, ASCII es realmente un subconjunto de Unicode. Debido a la gran cantidad de caracteres representables con Unicode resulta imposible generar una tabla con todos ellos. Sin embargo, podemos realizar consultas a través de la siguiente web:

<https://unicode-table.com/es/>

Para insertar un carácter Unicode en JAVA debemos hacerlo anteponiendo `\u` a la cifra que representa dicho carácter. Como las variables de tipo char utilizan 2 bytes, las podemos representar con 4 dígitos hexadecimales. A continuación se muestran una serie de ejemplos:

```
char letra1 = '\u03c0'; //carácter → π
char letra2 = '\u00a9'; //carácter → ©
char letra3 = '\u00A9'; //carácter → ©
char letra4 = '\u00ae'; //carácter → ®
char letra5 = '\u2605'; //carácter → ★
char letra6 = '\u2661'; //carácter → ♥
char letra7 = '\u221e'; //carácter → ∞
```

Ejercicios declaración de variables en JAVA

Indica las declaraciones
válidas en cada caso:

- a) `int base = 4; altura = 6;`
- b) `char letra-1, letra-2;`
- c) `boolean realizado = "false";`
- d) `boolean realizado = true;`
- e) `int num = 10.5;`
- f) `int 2partes = 5;`

- a) `int area = 4, perimetro;`
- b) `char letra_1, letra_2;`
- c) `boolean realizado = FALSE;`
- d) `boolean realizado = true`
- e) `float f2 = 10.5;`
- f) `double d2 = 10.002D;`
- g) `byte num = 128;`

Cadenas de texto (String)

Cadenas de texto en JAVA

Una cadena de texto (en inglés, "**String**") es un bloque de caracteres alfanuméricos, que usaremos para poder almacenar palabras y frases. Puede tener cero o más caracteres.

El valor de una variable de tipo String siempre debe ir entre comillas dobles.

Podemos "**concatenar**" cadenas (juntar dos cadenas para dar lugar a una nueva) con **el signo +**, igual que sumamos números. Por otra parte, la clase String tiene una gran cantidad de funciones (operaciones con nombre) que podemos aplicar para trabajar con ellas.

```
String nada = ""; //cadena vacía
String nombre = "Pepe";
String apellido = "Martínez";
String nombreCompleto1 = nombre + apellido; //PepeMartínez
String nombreCompleto2 = nombre + " " + apellido; //Pepe Martínez
```

Combinando Strings con variables numéricas

Podemos combinar Strings tanto con cantidades numéricas, como con variables que las contienen. Sin embargo, debemos tener en cuenta que si queremos **mostrar el cálculo** dentro del String, dicho cálculo **deberemos ponerlo entre paréntesis**, de lo contrario, lo **concatenará**.

```
String nombre = "Pepe";  
String frase;  
int cantidad = 20;  
frase = nombre + " tiene " + cantidad + " años";  
frase = nombre + " tiene " + cantidad + 5 + " años";  
frase = nombre + " tiene " + (cantidad + 5) + " años";  
frase = nombre + " tiene " + 15 + 5 + " años";  
frase = nombre + " tiene " + (15 + 5) + " años";  
frase = nombre + " tiene " + (15 - 5) + " años";  
frase = nombre + " tiene " + 15 - 5 + " años";  
frase = 15 + 5 + nombre;  
frase = 5 - 15 + nombre;  
frase = 5 - 10 + nombre + 5 + 10 + (5 + 5);
```

Combinando Strings con variables numéricas

Podemos combinar Strings tanto con cantidades numéricas, como con variables que las contienen. Sin embargo, debemos tener en cuenta que si queremos **mostrar el cálculo** dentro del String, dicho cálculo **deberemos ponerlo entre paréntesis**, de lo contrario, lo **concatenará**.

```
String nombre = "Pepe";
String frase;
int cantidad = 20;
frase = nombre + " tiene " + cantidad + " años";           //Pepe tiene 20 años
frase = nombre + " tiene " + cantidad + 5 + " años";       //Pepe tiene 205 años
frase = nombre + " tiene " + (cantidad + 5) + " años";     //Pepe tiene 25 años
frase = nombre + " tiene " + 15 + 5 + " años";             //Pepe tiene 155 años
frase = nombre + " tiene " + (15 + 5) + " años";           //Pepe tiene 20 años
frase = nombre + " tiene " + (15 - 5) + " años";           //Pepe tiene 10 años
frase = nombre + " tiene " + 15 - 5 + " años";             //ERROR
frase = 15 + 5 + nombre;                                    //20Pepe
frase = 5 - 15 + nombre;                                    //-10Pepe
frase = 5 - 10 + nombre + 5 + 10 + (5 + 5);               //-5Pepe51010
```

Combinando Strings con variables numéricas

```
int cantidad = 10;
int precio = 50;
float num = 2.5f;
float num2 = 10f;
String s1 = "El total es "+(cantidad+precio);
String s2 = "El total es "+cantidad*precio;
String s3 = "El total es "+cantidad+precio;
String s4 = cantidad+precio;
String s5 = cantidad*precio;
String s6 = (cantidad+precio);
String s7 = ""+cantidad+precio;
String s8 = ""+(cantidad+precio);
String s9 = "El total es "+(cantidad+num);
String s10 = "El total es "+(cantidad/3);
String s11 = "El total es "+(num2/3);
String s12 = "El total es "+(cantidad/3f);
String s13 = "El total es "+(cantidad/3d);
```


Combinando Strings con variables numéricas

```
int cantidad = 10;
int precio = 50;
float num = 2.5f;
float num2 = 10f;
String s1 = "El total es "+(cantidad+precio); //El total es 60
String s2 = "El total es "+cantidad*precio; //El total es 500
String s3 = "El total es "+cantidad+precio; //El total es 1050
String s4 = cantidad+precio; //ERROR
String s5 = cantidad*precio; //ERROR
String s6 = (cantidad+precio); //ERROR
String s7 = ""+cantidad+precio; //1050
String s8 = ""+(cantidad+precio); //60
String s9 = "El total es "+(cantidad+num); //El total es 12.5
String s10 = "El total es "+(cantidad/3); //El total es 3
String s11 = "El total es "+(num2/3); //El total es 3.3333333
String s12 = "El total es "+(cantidad/3f); //El total es 3.3333333
String s13 = "El total es "+(cantidad/3d); //El total es 3.3333333333333335
```

Cadenas de texto en JAVA

En **ningún momento** estamos **modificando** el **String de partida**. Eso sí, en muchos de los casos creamos un String modificado a partir del original.

El método "**compareTo()**" se basa en el orden lexicográfico: una cadena que empiece por "A" se considerará "menor" que otra que empiece por la letra "B"; si la primera letra es igual en ambas cadenas, se pasa a comparar la segunda, y así sucesivamente. Las mayúsculas y minúsculas se consideran diferentes.

Un comentario extra sobre los Strings: **Java convertirá a String todo aquello que indiquemos entre comillas dobles**. Así, son válidas expresiones como "Prueba".length() y también podemos concatenar varias expresiones dentro de una orden `System.out.println()`:

Cadenas de texto en JAVA

A continuación se muestran **algunos de los métodos más utilizados** cuando trabajamos con Strings:

Método	Descripción
<code>length()</code>	Devuelve la longitud (número de caracteres) de la cadena
<code>charAt(int pos)</code>	Devuelve el carácter que hay en cierta posición
<code>toLowerCase()</code>	Devuelve la cadena convertida a minúsculas
<code>toUpperCase()</code>	Devuelve la cadena convertida a mayúsculas
<code>substring(int desde, int hasta)</code>	Devuelve una subcadena: varias letras a partir de una posición dada
<code>replace(char antiguo, char nuevo)</code>	Devuelve una cadena con un carácter reemplazado por otro
<code>trim()</code>	Devuelve una cadena sin espacios blanco iniciales ni finales
<code>startsWith(String subcadena)</code>	Indica si la cadena empieza con una cierta subcadena
<code>endsWith(String subcadena)</code>	Indica si una cadena termina con una cierta subcadena

Cadenas de texto en JAVA

Método	Descripción
<code>indexOf(String subcadena, [int desde])</code>	Indica la posición en que se encuentra una cierta subcadena (buscando desde el principio a partir de una posición opcional)
<code>lastIndexOf(String subcadena, [int desde])</code>	Indica la posición en que se encuentra una cierta subcadena (buscando desde el final a partir de una posición opcional)
<code>valueOf(objeto)</code>	Devuelve un String que es la representación como texto del objeto que se le indique (número, boolean, etc.)
<code>concat(String cadena)</code>	Devuelve la cadena con otra añadida al final
<code>equals(String cadena)</code>	Mira si dos cadena son iguales (lo mismo que ==)
<code>equalsIgnoreCase(String cadena)</code>	Mira si dos cadena son iguales, pero despreciando las diferencias entre mayúsculas y minúsculas
<code>compareTo(String cadena2)</code>	Compara una cadena con la otra (devuelve 0 si son iguales, negativo si la cadena es menor que cadena2 y positivo si es mayor)

Ejemplos de uso de funciones en Strings

A continuación vemos una serie de ejemplos en el uso de funciones de tipo cadena

```
String nombre = "Pepe";
String apellidos = "Martínez García";
String nombreCompleto = nombre + apellidos;
int longitud = nombre.length();
int longitud2 = "245".length();
char letra = apellidos.charAt(3);
String cadena1 = nombreCompleto.substring(0,4);
String cadena2 = " Hola ".trim();
String cadena3_1 = cadena1.substring(3,4);
String cadena3_2 = cadena1.substring(3,3);
String cadena3_3 = cadena1.substring(4,3);
String cadena4 = (cadena2 + "Hola").toLowerCase();
int posicion1 = cadena4.indexOf("o");
int posicion2 = cadena4.indexOf("hola");
int posicion3 = cadena4.indexOf("Hola");
```

Ejemplos de uso de funciones en Strings

A continuación vemos una serie de ejemplos en el uso de funciones de tipo cadena

```
String nombre = "Pepe"; //Pepe
String apellidos = "Martínez García"; //Martínez García
String nombreCompleto = nombre + apellidos; //PepeMartínez García
int longitud = nombre.length(); //4
int longitud2 = "245".length(); //3
char letra = apellidos.charAt(3); //t
String cadena1 = nombreCompleto.substring(0,4); //Pepe
String cadena2 = " Hola ".trim(); //Hola
String cadena3_1 = cadena1.substring(3,4); //e
String cadena3_2 = cadena1.substring(3,3); //
String cadena3_3 = cadena1.substring(4,3); //ERROR
String cadena4 = (cadena2 + "Hola").toLowerCase(); //holahola
int posicion1 = cadena4.indexOf("o"); //1
int posicion2 = cadena4.indexOf("hola"); //0
int posicion3 = cadena4.indexOf("Hola"); //-1
```

Secuencias de escape

<code>\n</code>	Salto de línea. Sitúa el cursor al principio de la línea siguiente
<code>\b</code>	Retroceso. Mueve el cursor un carácter atrás en la línea actual.
<code>\t</code>	Tabulador horizontal. Mueve el cursor hacia adelante a una distancia determinada por el tabulador.
<code>\r</code>	Ir al principio de la línea. Mueve el cursor al principio de la línea actual.
<code>\f</code>	Nueva página. Mueve el cursor al principio de la siguiente página.
<code>\"</code>	Comillas. Permite mostrar por pantalla el carácter de las comillas dobles.
<code>\'</code>	Comilla simple. Permite mostrar por pantalla el carácter de la comilla simple.
<code>\\</code>	Barra inversa.
<code>\u</code>	Carácter Unicode, representa un dígito hexadecimal del carácter Unicode.

Ejemplos de secuencias de escape

A continuación veamos una serie de ejemplos en el uso de secuencias de escape

```
String nombre = "Pepe";
String apellidos = "Martínez García";
String nombreCompleto1 = nombre+" "+apellidos;           // Pepe Martínez García
String nombreCompleto2 = "\""+nombre+apellidos+"\"";      // "+nombre+apellidos+"
String nombreCompleto3 = "\""+nombre+apellidos+"\"";      // "PepeMartínez García"
String nombre2 = nombre+"\b"+nombre;                      // PepPepe
String apellidos2 = apellidos+"\r,"+nombre;               // ,Pepeñez García
String apellidos3 = apellidos+"\b,"+nombre;               // Martínez Garcí,Pepe
String apellidos4 = apellidos+"\\,"+nombre;               // Martínez García\\,Pepe
String apellidos5 = "\\u00A9 "+nombre;                    // © Pepe
String apellidos6 = "\\u00AE "+nombre;                    // ® Pepe
String apellidos7 = "\\u2605 "+nombre;                    // ★ Pepe
String apellidos8 = "\\u2661 "+nombre;                    // ♥ Pepe
```


Bloques de texto

A partir de la versión 15 de JAVA están disponibles los **text-blocks**, esto **nos permite introducir Strings de varias líneas** de forma sencilla. Para ello tendremos que incluir el texto entre **comillas dobles tres veces seguidas**, tanto para abrir, como para cerrar la String. Además, **no debemos escribir nada en las líneas donde van las comillas**

```
String nueva =
```

```
"""
```

```
Esto es un String multilínea
```

```
Estoy haciendo una prueba
```

```
....
```

```
""";
```

String con 3 líneas de texto

```
String nueva = """
```

```
Esto es un String multilínea
```

```
Estoy haciendo una prueba
```

```
....
```

```
""";
```

String con 4 líneas de texto

Ejercicio de Strings en JAVA

```
String texto1 = "Hola"; // Forma "sencilla"
String texto2 = new String("Prueba"); // Usando un "constructor"
System.out.println("La primera cadena de texto es :");
System.out.println(texto1);
System.out.println("Concatenamos las dos: " + texto1 + texto2);
System.out.println("Concatenamos varios: " + texto1 + 5 + " " + 23.5);
System.out.println("La longitud de la segunda es: " + texto2.length());
System.out.println("La segunda letra de texto2 es: " + texto2.charAt(1));
System.out.println("La cadena texto2 en mayúsculas: " + texto2.toUpperCase());
System.out.println("Tres letras desde la posición 1: " + texto2.substring(1,3));
```

Solución ejercicio Strings en JAVA

El resultado tras ejecutar el programa anterior sería el siguiente:

La primera cadena de texto es :

Hola

Concatenamos las dos: HolaPrueba

Concatenamos varios: Hola5 23.5

La longitud de la segunda es: 6

La segunda letra de texto2 es: r

La cadena texto2 en mayúsculas: PRUEBA

Tres letras desde la posición 1: ru

Recomendaciones al declarar variables

Recomendaciones al declarar variables en JAVA

Una buena elección del identificador de una variable puede ser difícil incluso para programadores experimentados, sobre todo si tenemos muchas variables. Aunque como veremos más adelante, tampoco es recomendable tener muchas variables. Esto lo entenderemos mejor cuando empecemos con la programación orientada a objetos.

Algo muy importante, **debemos elegir nombres que sean significativos, no pasa nada si los identificadores** de las variables en ocasiones **son largos**, siempre y cuando mejoren la legibilidad del código.

No empezar con mayúsculas, ya que es una forma de diferenciar los objetos de las variables. Si el identificador está formado por varias palabras, se deben separar con mayúsculas o con guión bajo.

Ejemplos de declaración de variables en JAVA

NO SE RECOMIENDA

```
int CANTIDAD = 3;  
String Canal = "Aulaenlanube";  
String nombrecompleto = "Marta García";  
String j = "Jose Pérez";
```

RECOMENDADO

```
int cantidad = 3;  
String canal = "Aulaenlanube";  
String nombreCompleto = "Marta García";  
String nombre_completo = "Jose Pérez";
```

Constantes

Constantes

Son valores fijos. Java soporta constantes con nombres, y se crean de la siguiente forma:

```
final float PI = 3.14f;  
final String NOMBRE_CANAL = "Aulaenlanube";
```

La palabra clave **final** es la que hace que **PI** sea un valor que no pueda ser modificado, y **NOMBRE_CANAL** tampoco.

*Aunque **no es obligatorio**, es muy recomendable y una buena práctica **declarar los nombres de las constantes completamente en mayúsculas**, es una forma de diferenciarlas del resto de variables.*

Valores por defecto de las variables en JAVA

Si no especificamos un valor al declarar una variable, internamente JAVA asigna un **valor por defecto** según el tipo.

Aunque en principio **no podemos operar** con variables que no tienen asignada ningún valor. Más adelante veremos que hay ocasiones donde esta información puede resultarnos útil, como por ejemplo con los cuando trabajamos con Arrays.

```
byte → 0
short → 0
int → 0
long → 0
float → 0.0f
double → 0.0d
boolean → false
char → '\u0000'
String → null
```


Comentarios

Comentarios en JAVA

- Los comentarios pueden ser colocados en cualquier parte de nuestro código en Java. Su objetivo es **dar orden** al código y **hacerlo más comprensible**, en especial cuando **otras personas** tienen que leerlo y trabajar sobre él.
- Los **comentarios** son código JAVA que **no se ejecuta**.
- Aunque cuando empezamos a programar es muy habitual escribir muchos comentarios sobre nuestro código, **debemos evitar el abuso** de los mismos.
- En JAVA hay 3 formas distintas de añadir comentarios al código
 - **Comentarios de una línea**
 - **Comentarios multilínea**
 - **Comentarios de documentación**

Comentarios en JAVA

```
//Comentarios de una línea, empiezan con doble barra
```

```
/*  
Comentario multilínea  
Esto es una segunda línea de comentario  
*/
```

```
/**  
Comentario de documentación, empiezan con 2 asteriscos. Este  
tipo de comentarios los utiliza la herramienta javadoc  
*/
```

Operadores

Operaciones matemáticas básicas

Podemos utilizar las distintas operaciones matemáticas para utilizarlas en expresiones e incluso combinarlas con variables. Pero claro, debemos tener en cuenta el tipo de variable cuando realizamos estas operaciones. Podemos sumar enteros e incluso Strings para concatenarlas, pero no podemos multiplicar dos Strings, ni sumar booleanos con char, ni restar Strings, etc.

Operador	Operación	Operador	Operación
+	Suma o signo	+=	Suma y asignación
-	Resta o signo	-=	Resta y asignación
*	Multiplicación	*=	Multiplicación y asignación
/	División	/=	División y asignación
%	Módulo	%=	Módulo y asignación
++	Incremento en 1	--	Decremento en 1

Ejemplos operaciones matemáticas

```
int a = 3, b = 2;
```

```
float f1 = 3f, f2 = 2f;
```

```
System.out.println("3 + 2 = "+a+b);
```

```
System.out.println("3 + 2 = "+(a+b));
```

```
System.out.println(a+" + "+b+" = "+(a+b));
```

```
System.out.println(a+" - "+b+" = "+(a-b));
```

```
System.out.println(a+" x "+b+" = "+(a*b));
```

```
System.out.println(a+" / "+b+" = "+(a/b));
```

```
System.out.println(f1+" / "+f2+" = "+(f1/f2));
```

```
System.out.println(a+" / "+f2+" = "+(a/f2));
```

```
System.out.println(b+" % "+a+" = "+(b%a));
```

```
System.out.println(f2+" % "+f1+" = "+(f2%f1));
```

Ejemplos operaciones matemáticas

```
int a = 3, b = 2;
```

```
float f1 = 3f, f2 = 2f;
```

```
System.out.println("3 + 2 = "+a+b);           // 3 + 2 = 32
```

```
System.out.println("3 + 2 = "+(a+b));          // 3 + 2 = 5
```

```
System.out.println(a+" + "+b+" = "+(a+b));     // 3 + 2 = 5
```

```
System.out.println(a+" - "+b+" = "+(a-b));     // 3 - 2 = 1
```

```
System.out.println(a+" x "+b+" = "+(a*b));     // 3 x 2 = 6
```

```
System.out.println(a+" / "+b+" = "+(a/b));     // 3 / 2 = 1
```

```
System.out.println(f1+" / "+f2+" = "+(f1/f2)); // 3.0 / 2.0 = 1.5
```

```
System.out.println(a+" / "+f2+" = "+(a/f2));   // 3 / 2.0 = 1.5
```

```
System.out.println(b+" % "+a+" = "+(b%a));     // 2 % 3 = 2
```

```
System.out.println(f2+" % "+f1+" = "+(f2%f1)); // 2.0 % 3.0 = 2.0
```

Incrementos y asignaciones abreviadas

Hay varias operaciones muy habituales, que tienen una sintaxis abreviada en Java. Por ejemplo, para **sumar 2 a una variable "a"**, la forma "normal" de conseguirlo sería: **a = a + 2;** pero existe una **forma abreviada** en Java: **a += 2;**

Al igual que tenemos el operador **+=** para aumentar el valor de una variable, tenemos **-=** para disminuirlo, **/=** para dividirla y ***=** para multiplicarla. Por **ejemplo**, para **multiplicar por 10 el valor de la variable "b"** haríamos **b *= 10;**

También podemos **aumentar o disminuir** en una unidad el valor de una variable, empleando los operadores de "**incremento**" (**++**) y de "**decremento**" (**--**). Así, para sumar 1 al valor de "a", podemos emplear cualquiera de estas tres formas:

```
a = a+1;    //2
a += 1;     //3
a++;        //4
```


Incrementos y decremento

Los operadores de incremento y de decremento se pueden escribir antes o después de la variable. Así, es lo mismo escribir estas dos líneas:

```
a++;  
++a;
```

Pero hay una diferencia si ese valor se asigna a otra variable "al mismo tiempo" que se incrementa/decrementa:

```
int c = 5;  
int b = c++;
```

Guarda como resultado c = 6 y b = 5, se asigna el valor a "b" **antes** de incrementar "c".
Sin embargo:

```
int c = 5;  
int b = ++c;
```

Guarda como resultado c = 6 y b = 6, se asigna el valor a "b" **después** de incrementar "c").

Ejemplos operaciones matemáticas

//OPERADORES CON ASIGNACIÓN

```
int a = 3, b = 2, c = 1;
```

```
a += b;
```

```
c += a;
```

```
b += a + b;
```

```
c -= a;
```

```
c *= a / 2;
```

```
b %= 5;
```

```
b /= c;
```

```
b *= a + b;
```

```
a /= b - c * 5;
```

[illegible]

Ejemplos operaciones matemáticas

//OPERADORES CON ASIGNACIÓN

```
int a = 3, b = 2, c = 1;
```

```
a += b;           // a = a + b → 5
```

```
c += a;           // c = c + a → 6
```

```
b += a + b;       // b = b + (a + b) → 9
```

```
c -= a;           // c = c - a → 1
```

```
c *= a / 2;       // c = c * (a / 2) → 2
```

```
b %= 5;           // b = b % 5 → 4
```

```
b /= c;           // b = b / c → 2
```

```
b *= a + b;       // b = b * (a + b) → 14
```

```
a /= b - c * 5;   // a = a / (b - c * 5) → 1
```

a	b	c
3	2	1
5	2	1
5	2	6
5	9	6
5	9	1
5	9	2
5	4	2
5	2	2
5	14	2
1	14	2

Ejemplos incrementos y decrementos

Ejemplos de expresiones matemáticas combinadas:

```
int a=1, b=2, c=3;  
b = ++c;  
a += b++;  
a = a+++a;  
a -= b--;  
c = a++ - ++b;  
c -= ++a;  
a -= ++c;  
a -= c++;  
a -= --c;
```

Ejemplos incrementos y decrementos

Ejemplos de expresiones matemáticas combinadas:

```
int a=1, b=2, c=3;
b = ++c;           // b = (++c)
a += b++;          // a = a + b++
a = a+++a;         // a = (a++) + a
a -= b--;          // a = a - b--
c = a++ - ++b;     // c = a++ - (++b)
c -= ++a;          // c = c - (++a)
a -= ++c;          // a = a - (++c)
a -= c++;          // a = a - c++
a -= --c;          // a = a - (--c)
```

a	b	c
1	2	3
1	4	4
5	5	4
11	5	4
6	4	4
7	5	1
8	5	-7
14	5	-6
20	5	-5
26	5	-6

Ejemplos incrementos y decremento

```
int a = 1;  //En todas las instrucciones iniciamos a 1 una previamente
```

```
//-----
```

```
a = ++a;
```

```
a = --a;
```

```
a = a--;
```

```
a = a++;
```

```
a = a+a--;
```

```
a = a+a++;
```

```
//-----
```

```
a = a--+a--;
```

```
a = a--+a--+a--;
```

```
a = a--+a--+a--+a--;
```

```
a = a--+a--+a--+a--+a--;
```

```
a = a--+a--+a--+a--+a--+a--;
```

Ejemplos incrementos y decrementos

```
int a = 1;  //En todas las instrucciones iniciamos a 1 una previamente
//-----

a = ++a;    //guarda 2, todo ok
a = --a;    //guarda 0, todo ok
a = a--;    //1º(a=a→1) 2º(a--→0) 3º(a=1º→1)
a = a++;    //1º(a=a→1) 2º(a++→2) 3º(a=1º→1)
a = a+a--;  //1º(a+a→2) 2º(a--→0) 3º(a=1º→2)
a = a+a++;  //1º(a+a→2) 2º(a++→2) 3º(a=1º→2)
//-----

a = a--+a--; // 1º(a→-1) 2º(1+0→1)
a = a--+a--+a--; // 1º(a→-2) 2º(1+0+(-1)→0)
a = a--+a--+a--+a--; // 1º(a→-3) 2º(1+0+(-1)+(-2)→-2)
a = a--+a--+a--+a--+a--; // 1º(a→-4) 2º(1+0+(-1)+(-2)+(-3)→-5)
a = a--+a--+a--+a--+a--+a--; // 1º(a→-5) 2º(1+0+(-1)+(-2)+(-3)+(-4)→-9)
```

Ejemplos char con incrementos y decrementos

A continuación se muestran una serie ejemplos de **incrementos y decrementos** cuando utilizamos variables de **tipo char**. En JAVA las variables char pueden operar con enteros, y el resultado de la operación representa su valor en la tabla [ASCII](#).

```
char letra = 'A';  
letra++;      //B  
letra+=5;     //G  
letra--;      //F  
letra-=2;     //D  
letra = letra + 2;  //ERROR  
letra = (char)(letra + 2); //F  
letra = (char)(letra + '1'); // F(70)+1(49) = w(119)
```


Expresiones lógicas

Operadores relacionales

En JAVA, al igual que en pseudocódigo, también tenemos las **expresiones lógicas**.

En este tipo de expresiones el resultado será de tipo booleano. Es decir, verdadero o falso.

Dichas expresiones están formadas por:

- Operandos: constantes, variables y expresiones lógicas.
- Operadores: **relacionales** (**==**, **<**, **>**, **<=**, **>=**, **!=**).

Además podemos asignar el resultado de una expresión lógica a una variable de tipo **boolean**

Operador	Operación
==	Igual
!=	Distinto
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

```
int a = 0, b = 1;  
boolean iguales = a == b; //false
```

Asignando booleanos con operadores relacionales

Ejemplos de asignación sobre variables boolean:

```
int precio1 = 10;
int precio2 = 30;
boolean barato = precio1 > precio2;           //false
boolean iguales1 = 40 == (precio1 + precio2); //true
boolean iguales2 = 40 == precio1 + precio2;   //true
boolean iguales2 = (40 == precio1) + precio2; //ERROR
boolean distintos = precio1 != precio2;       //true
```

Para **comparar Strings** se utiliza equals

```
String nombre1 = "Pepe";
String nombre2 = "Jose";
String nombre3 = "pepe"
boolean iguales1 = nombre1.equals(nombre2);   //false
boolean iguales2 = nombre1.equals(nombre3);   //false
boolean iguales3 = nombre1.equalsIgnoreCase(nombre3); //true
```

Ejemplos operadores relacionales

Ejemplos de asignación sobre variables boolean:

```
boolean iguales = false;  
int precio1 = 1, precio2 = 3;  
float precio1f = 1f, precio2f = 3f;  
double precio1d = 1d, precio2d = 3d;
```

```
iguales = precio1 == precio1f;  
iguales = precio1 == precio1d;  
iguales = precio1f == precio1d;  
iguales = precio2 == 2.0 + 1.0;  
iguales = precio1/10 == 0.1;  
iguales = precio1/10f == 0.1;  
iguales = precio1/10d == 0.1;  
iguales = precio1/10 < 0.1;
```

```
boolean iguales = false;  
float nuevo = 2/3;  
iguales = nuevo == 2/3;  
iguales = nuevo == 2f/3f;
```

```
nuevo = 2f/3f;  
iguales = nuevo == 2/3;  
iguales = nuevo == 2f/3f;  
iguales = nuevo == 0.6666667f;  
iguales = nuevo == 0.6666666f;  
iguales = nuevo == 0.66666666f;  
iguales = nuevo == 0.66666666d;  
iguales = nuevo == 0.6666666666666666d;  
iguales = 2d/3d == 0.6666666666666666d;
```

Ejemplos operadores relacionales

Ejemplos de asignación sobre variables boolean:

```
boolean iguales = false;
int precio1 = 1, precio2 = 3;
float precio1f = 1f, precio2f = 3f;
double precio1d = 1d, precio2d = 3d;

iguales = precio1 == precio1f; //true
iguales = precio1 == precio1d; //true
iguales = precio1f == precio1d; //true
iguales = precio2 == 2.0 + 1.0; //true
iguales = precio1/10 == 0.1; //false
iguales = precio1/10f == 0.1; //false
iguales = precio1/10d == 0.1; //true
iguales = precio1/10 < 0.1; //true
```

```
boolean iguales = false;
float nuevo = 2/3; //0
iguales = nuevo == 2/3; //true
iguales = nuevo == 2f/3f; //false

nuevo = 2f/3f; //0.6666667
iguales = nuevo == 2/3; //false
iguales = nuevo == 2f/3f; //true
iguales = nuevo == 0.6666667f; //true
iguales = nuevo == 0.6666666f; //false
iguales = nuevo == 0.66666666f; //true
iguales = nuevo == 0.66666666d; //false
iguales = nuevo == 0.6666666666666666d; //false
iguales = 2d/3d == 0.6666666666666666d; //true
```

Operadores lógicos

Podremos enlazar varias condiciones, para indicar qué hacer cuando se cumplan ambas, o sólo una de ellas, o cuando una no se cumpla. Los operadores que nos permitirán enlazar condiciones son:

Operador	Operación	Significado
!	NOT	Negación lógica
&&	AND	Conjunción lógica (Y lógico)
	OR	Disyunción (O lógico)
^	XOR	Disyunción Exclusiva (XOR)

Por ejemplo, la forma de decir "**si a vale 3 y b es mayor que 5, o bien, si a vale 7 y b no es menor que 4**" en sintaxis JAVA sería:

```
if ((a == 3 && b > 5) || (a == 7 && !(b < 4)))
```

```
if (a == 3 && b > 5 || a == 7 && !(b < 4))
```

Asignando booleanos con operadores lógicos

Ejemplos de asignación sobre variables boolean:

```
int precio1 = 10;
int precio2 = 20;
int precio3 = 30;

boolean caro = precio1 >= 30;           //false
boolean barato = precio1 >= 0 && precio1 <= 10; //true
boolean imposible = precio1 >= 10 && precio1 < 10; //false
boolean iguales = precio1 == precio2 && precio2 == precio3; //false
boolean distintos = precio1 != precio2 && precio1 != precio3 && precio3 != precio2; //true
boolean max1 = precio1 >= precio2 && precio1 >= precio3; //false
boolean max2 = precio2 >= precio1 && precio2 >= precio3; //false
boolean max3 = precio3 >= precio1 && precio3 >= precio2; //true
boolean algunoCaro = precio1 >= 30 || precio2 >= 30 || precio3 >= 30; //true
boolean algunoDistinto = precio1 != precio3 || precio3 != precio2; //true
boolean combinada = precio1 <= precio2 || precio1 == precio2 && precio1 == precio3; //true
```

Se debe tener en cuenta que el **AND** (&&) siempre tiene prioridad frente al **OR** (||)

Asignando booleanos con operadores lógicos

Ejemplos de asignación sobre variables boolean:

```
int a = 1, b = 2;
boolean condicion1 = true;
boolean condicion2 = false;
boolean resultado;
resultado = condicion1 && condicion2;
resultado = condicion1 || condicion2;
resultado = condicion1 && a < b;
resultado = condicion2 && a < b;
resultado = condicion1 && !condicion2;
resultado = !(condicion1 && !condicion2);
resultado = !(condicion1 && condicion2);
resultado = condicion1 ^ condicion2;
resultado = condicion1 ^ a != b;
resultado = !condicion1 ^ !(a != b);
resultado = !(condicion1 ^ a != b);
```


Asignando booleanos con operadores lógicos

Ejemplos de asignación sobre variables boolean:

```
int a = 1, b = 2;
boolean condicion1 = true;
boolean condicion2 = false;
boolean resultado;
resultado = condicion1 && condicion2;           //false
resultado = condicion1 || condicion2;           //true
resultado = condicion1 && a < b;                 //true
resultado = condicion2 && a < b;                 //false
resultado = condicion1 && !condicion2;           //true
resultado = !(condicion1 && !condicion2);        //false
resultado = !(condicion1 && condicion2);         //true
resultado = condicion1 ^ condicion2;            //true
resultado = condicion1 ^ a != b;                //false
resultado = !condicion1 ^ !(a != b);            //false
resultado = !(condicion1 ^ a != b);             //true
```

Operadores a nivel de bits

Operadores a nivel de bits

En JAVA se permite la manipulación **bit a bit** con cualquiera de los tipos enteros (**byte, short, int, long y char**). Las distintas operaciones que podemos realizar las tenemos en la siguiente tabla:

Cabe recordar que podemos declarar el valor de una variable entera tanto en hexadecimal como en binario.

```
int a = 3;  
int b = 0b11;  
int c = 0x3;
```

Operador	Operación	Significado
~	NOT	Negación lógica
&	AND	Conjunción lógica (Y lógico)
	OR	Disyunción (O lógico)
^	XOR	Disyunción Exclusiva (XOR)
>>	a >> desp	Desplaza 'a', 'desp' bits a la derecha
<<	a << desp	Desplaza 'a', 'desp' bits a la izquierda
>>>	a >>> desp	Desplaza 'a', 'desp' bit a la derecha, sin signo

Operadores a nivel de bits

```
int a = 3;  //00000011
int b = 4;  //00000100
int c = 8;  //00001000
int d = 15; //00001111
int resultado = 0;

resultado = a & b;      //00000000 → 0
resultado = a | b;      //00000111 → 7
resultado = a ^ d;      //00001100 → 12
resultado = ~a;         //11111100 → -4
resultado = a | b | c;  //00001111 → 15
resultado = a & b & c;  //00000000 → 0
resultado = d >> 1;     //00000111 → 7
resultado = d << 1;     //00011110 → 30
resultado = d >> 4;     //00000000 → 0
resultado = d << 4;     //11110000 → 240
byte e = (byte)(d << 4); //11110000 → -16
```

Operadores a nivel de bits

Para el desplazamiento a la derecha JAVA proporciona dos operadores: `>>` y `>>>`. **El primero** es usado en el desplazamiento a la derecha **con signo**, mientras que **el segundo** se usa en el desplazamiento a la derecha **sin signo**. La diferencia es que el **primero tiene en cuenta el signo del número**, y según este signo **se rellenará con un 1(para negativos) o con un 0(para positivos)**.

```
int a = -1;    //11111111 11111111 11111111 11111111
int b = -16;   //11111111 11111111 11111111 11110000
int resultado = 0;

resultado = a >> 1;    //11111111 11111111 11111111 11111111 → -1
resultado = a >>> 1;   //01111111 11111111 11111111 11111111 → 2147483647
resultado = b >> 1;    //11111111 11111111 11111111 11111000 → -8
resultado = b >>> 1;   //01111111 11111111 11111111 11111000 → 2147483640
resultado = b >> 24;   //11111111 11111111 11111111 11111111 → -1
resultado = b >>> 24;  //00000000 00000000 00000000 11111111 → 255
a >>= 1;            //11111111 11111111 11111111 11111111 → -1
a >>>= 1;           //01111111 11111111 11111111 11111111 → 2147483647
```

Precedencia de operadores

Precedencia de los operadores

Al igual que en las matemáticas, en la programación, es muy importante el orden de precedencia que tienen los operadores, y así poder obtener correctamente el resultado de un algoritmo. Por ello en la siguiente tabla se muestra la precedencia de los distintos operadores separada en grupos de más prioridad (grupo 1), a menos nivel de prioridad (grupo 14).

Grupo 1	()
Grupo 2	+ - ++ -- ~ !
Grupo 3	* / %
Grupo 4	+ -
Grupo 5	<< >> >>>
Grupo 6	> >= < <=
Grupo 7	== !=
Grupo 8	&
Grupo 9	^
Grupo 10	
Grupo 11	&&
Grupo 12	
Grupo 13	?:
Grupo 14	= += -= *= /= %= &= = ^= <<= >>= >>>=

Ejemplos de precedencia de operadores

Ejemplos de precedencia de operadores.

```
int n = 2+3/2*8/4+5;
```

```
n *= 4+2*3;
```

```
n += n+++3-n;
```

```
//-----
```

```
double d1 = 1, d2 = 2, d3;
```

```
d3 = d2+++d1/d2*3+d1;
```

```
//-----
```

```
boolean cond1 = true, cond2 = false, cond3 = false;
```

```
int a = 1, b = 2, c = 3;
```

```
cond3 = a++ == b || a == b;
```

```
cond3 = a++ == c || b != --c && cond1 ^ cond3;
```

```
cond3 = !(!cond2) || cond1 && !cond3 ^ cond3 && cond2;
```


Ejemplos de precedencia de operadores

Ejemplos de precedencia de operadores.

```
int n = 2+3/2*8/4+5; // n = 2+(3/2*8/4)+5 → 9
n *= 4+2*3;         // n = n * (4+2*3) → 90
n += n+++3-n;       // n = n + (n++) + 3 - n → 92
//-----

double d1 = 1, d2 = 2, d3;
d3 = d2+++d1/d2*3+d1; // d3 = (d2++) + (d1/d2*3) + d1 → 4.0
//-----

boolean cond1 = true, cond2 = false, cond3 = false;
int a = 1, b = 2, c = 3;
cond3 = a++ == b || a == b; //true
cond3 = a++ == c || b != --c && cond1 ^ cond3; //false
cond3 = !(!cond2) || cond1 && !cond3 ^ cond3 && cond2; //false
```

Clase Math

Funciones matemáticas

En JAVA existe una clase que nos facilita la realización de funciones matemáticas complejas, dicha **clase se llama Math**. Para utilizar las funciones de dicha clase **debemos escribir la palabra Math, luego un punto, y finalmente la función** que queramos utilizar. A continuación se muestran algunas de las funciones y constantes que incluye esta clase:

Método	Descripción
<code>round(double a)</code>	Devuelve el número completo más cercano.
<code>floor(double a)</code>	Devuelve el número completo más cercano por debajo.
<code>ceil(double a)</code>	Devuelve el número completo más cercano por arriba.
<code>sqrt(double a)</code>	Devuelve la raíz cuadrada de 'a'.
<code>pow(double a, double b)</code>	Devuelve 'a' elevado a 'b'.
<code>abs(x)</code>	Devuelve el valor absoluto de 'x'.
<code>log(double a)</code>	Devuelve el logaritmo natural de 'a'.

Funciones matemáticas

Método	Descripción
<code>random()</code>	Devuelve un número aleatorio entre 0 y 1.
<code>sin(double a)</code>	Devuelve el seno de un ángulo 'a' en radianes.
<code>cos(double a)</code>	Devuelve el coseno de un ángulo 'a' en radianes.
<code>tan(double a)</code>	Devuelve la tangente de un ángulo 'a' en radianes.
<code>asin(double a)</code>	Devuelve el ángulo cuyo seno es 'a'.
<code>acos(double a)</code>	Devuelve el ángulo cuyo coseno es 'a'.
<code>atan(double a)</code>	Devuelve el ángulo cuya tangente es 'a'

Constante	Descripción
PI	Devuelve el número PI, (double)
E	Devuelve el número E, (double)

Utilizando API para funciones matemáticas

```
int n = 2;
System.out.println(Math.pow(n,8));           //256.0

double radio = 5, area;
area = Math.PI * radio * radio;              //78.53981633974483
area = Math.PI*Math.pow(radio, 2);           //78.53981633974483
System.out.println(Math.ceil(area));          //79.0
System.out.println(Math.floor(area));         //78.0
System.out.println(Math.round(area));         //79
System.out.println(Math.sqrt(area*area));     //78.53981633974483

System.out.println(Math.sqrt(Math.pow(n,4))); //4.0
```

Decimales y precisión en las operaciones

Almacenamiento de decimales

Para almacenar decimales(números reales) JAVA nos proporciona dos tipos de variables, **float** y **double**. La diferencia radica en la cantidad de bytes que utilizan para almacenar información.

- Float utiliza 4 bytes (simple precisión)
- Double utiliza 8 bytes (doble precisión)

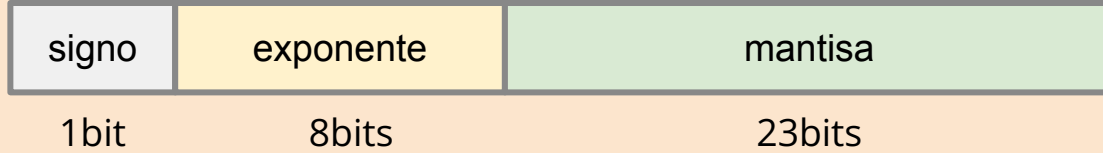
IEEE 754 es el estándar utilizado para la aritmética en coma flotante.

- Simple precisión utiliza 32 bits de base 2, se conoce oficialmente como **binary32**.
- Doble precisión utiliza 64 bits de base 2, se conoce oficialmente como **binary64**.

Todo ello implica que una variable de tipo double puede almacenar números más grandes y con mayor precisión.

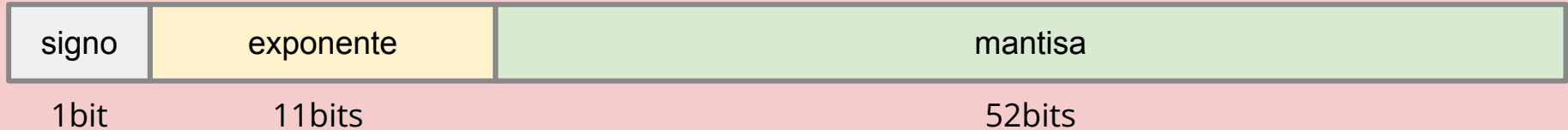
Almacenamiento de decimales

IEEE 754 DE SIMPLE PRECISIÓN (32bits)



El almacenamiento de simple precisión nos ofrece una precisión de 7 decimales.

IEEE 754 DE DOBLE PRECISIÓN (64bits)



El almacenamiento de doble precisión nos ofrece una precisión de 16 decimales.

Problemas de precisión

En computación, un número es almacenado en memoria en su forma binaria, una secuencia de bits, unos y ceros. Con números enteros no hay problema, pero con números decimales como 0.1 y 0.2, aunque parecen simples en el sistema decimal, son realmente números con infinitos decimales en su forma binaria. Veamos un ejemplo:

En decimal 0.5 es uno dividido por dos, un medio($\frac{1}{2}$). En decimal 0,5 se representa fácilmente, sin embargo al compararlo con otras divisiones, la cosa se complica.

Si dividimos uno entre tres, tenemos un tercio($\frac{1}{3}$), esto devuelve un número con infinitos decimales, 0.33333(periodo). Almacenar esta cantidad en sistema decimal resulta imposible ya que tenemos infinitos decimales, por lo tanto debemos asumir una pequeña pérdida de precisión al almacenar este tipo de información.

Problemas de precisión

Por la misma razón, en el sistema binario, en la división en potencias de 2, no hay pérdida de precisión y se almacena perfectamente. Sin embargo, al dividir entre diez nos devuelve $(1/10)$, qué almacenado en binario se convierte en un número con infinitos decimales.

En definitiva, en computación es imposible almacenar de forma exacta 0.1 y 0.2 usando el sistema binario. Del mismo modo, tampoco hay manera de guardar un tercio en fracción decimal sin perder precisión. Todo ello nos lleva a la siguiente pérdida de precisión.

```
System.out.println(0.1+0.2); //0.30000000000000004
```

También se pueden producir pérdidas de precisión si realizamos cálculos con las variables no adecuadas, por ejemplo

```
int a = 10;  
int b = 3;  
float c = a/b; //3.0
```

Entrada de datos

Datos introducidos por el usuario

En JAVA, tenemos la posibilidad de acceder a la entrada de teclado con la clase **Scanner**.

Con el método **".next()"** obtenemos los datos hasta llegar a un espacio en blanco. Sin embargo, el método **".nextLine()"** obtenemos los datos hasta encontrar un salto de línea.

```
import java.util.Scanner;

class MiScanner {
    public static void main(String args[]) {
        String nombre;
        System.out.print("Introduzca su nombre (una palabra): ");
        Scanner entrada = new Scanner(System.in); //creamos un Scanner
        nombre = entrada.next(); //leemos hasta llegar a un espacio en blanco
        System.out.println("Hola, " + nombre);
    }
}
```

Datos introducidos por el usuario

No sólo podemos leer cadenas de texto. Si lo siguiente que queremos leer es un número, podemos usar:

- `nombreVariableScanner.nextInt();`
- `nombreVariableScanner.nextFloat();`
- `nombreVariableScanner.nextDouble();`

Y si queremos obtener más de un dato, podemos repetir con "**.hasNext()**" (tiene siguiente), que nos devolverá un booleano (verdadero o falso).

Típicamente se usaría como parte de un bucle (instrucción repetitiva):

```
while (entrada.hasNext()) { ... }
```

Datos introducidos por el usuario

Cuando en un programa se leen por teclado datos numéricos y datos de tipo carácter o String debemos tener en cuenta que al introducir los datos y pulsar intro estamos también introduciendo en el buffer de entrada el intro.

Esto es, la instrucción:

```
int n = entrada.nextInt();
```

Asigna a 'n' el valor 5 pero el intro permanece en el buffer. Esto quiere decir que el buffer de entrada después de leer el entero tiene el carácter **\n**.

Por ejemplo, si ahora se pide que se introduzca por teclado una cadena de caracteres:

```
System.out.print("Introduzca su nombre: ");  
nombre = entrada.nextLine(); //lee una línea
```

Datos introducidos por el usuario

El método **nextLine()** extrae del buffer de entrada todos los caracteres hasta llegar a un intro y elimina el intro del buffer. En este caso se asigna una cadena vacía a la variable nombre y limpia el intro. Esto provoca que el programa no funcione correctamente, ya que no se detiene para que se introduzca el nombre.

Dado el siguiente ejemplo, ¿Qué ocurre?

```
import java.util.Scanner;
public class Cuadrado{
    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);
        String nombre;
        int n;
        System.out.print("Introduzca un número entero: ");
        n = entrada.nextInt();
        System.out.println("El cuadrado es: " + Math.pow(n,2));
        System.out.print("Introduzca su nombre: ");
        nombre = entrada.nextLine(); //guardamos una cadena vacía en nombre
        System.out.println("Hola " + nombre + "!!");
    }
}
```

Datos introducidos por el usuario

¿Cómo lo podemos solucionar?

```
public class Cuadrado{  
  
    public static void main(String[] args) {  
  
        Scanner entrada = new Scanner(System.in);  
        String nombre;  
        int n;  
        System.out.print("Introduzca un número entero: ");  
        n = entrada.nextInt();  
        entrada.nextLine();  
        System.out.println("El cuadrado es: " + Math.pow(n,2));  
        System.out.print("Introduzca su nombre: ");  
        nombre = entrada.nextLine(); //leemos correctamente la String  
        System.out.println("Hola " + nombre + "!!");  
    }  
}
```


Condiciones

Estructuras de control alternativas

Alternativas: se ejecuta una instrucción o conjunto de instrucciones después de evaluar una condición.

SIMPLE:

```
if (condición) {  
    instrucción1;  
    instrucción2;  
}  
instrucción3;
```

DOBLE:

```
if (condición) {  
    instrucción1;  
    instrucción2;  
} else {  
    instrucción3;  
}  
instrucción4;
```

ANIDADA:

```
if (condición1) {  
    instrucción1;  
    instrucción2;  
} else {  
    if (condición2) {  
        instrucción3;  
        instrucción4;  
    } else {  
        instrucción5;  
    }  
}  
instrucción6;
```

Condiciones (IF)

En cualquier lenguaje de programación es habitual tener que comprobar si se cumple una cierta condición. La forma "normal" de conseguirlo es empleando una construcción que recuerda a:

```
SI condición_a_comprobar ENTONCES  
    pasos_a_dar
```

En el caso de JAVA, la forma exacta será empleando **if** (si, en inglés), **seguido por la condición entre paréntesis**, e indicando finalmente entre llaves(opcionales) los pasos que queremos dar si se cumple la condición, así :

```
if (condición) { //la condición debe devolver un boolean  
    sentencia1;  
    sentencia2;  
    ...  
    sentenciaN;  
}
```

Condiciones (IF)

Por ejemplo:

```
if(x == 3) {  
    System.out.println("El valor es correcto");  
    resultado = 5;  
}
```

Nota: Si sólo queremos dar un paso en caso de que se cumpla la condición, no es necesario emplear llaves (aunque puede ser recomendable usar siempre las llaves, para no olvidarlas si más adelante ampliamos ese fragmento del programa). Las llaves serán imprescindibles sólo cuando se debe ejecutar más de una sentencia:

```
if(x == 3)  
    System.out.println("El valor es correcto");
```

Condiciones (ELSE)

El **'else'** siempre es opcional y nos permite indicar qué hacer en caso de que no se cumpla la condición. Al igual que el if, debe ir entre llaves si queremos ejecutar más de una instrucción.

Sería algo parecido a:

```
SI condición_a_comprobar ENTONCES
    pasos_a_dar
SINO
    pasos_alternativos
```

que en JAVA escribiríamos así:

```
if(condición) {
    sentencias1
}
else {
    sentencias2
}
```

```
if(x == 3) {
    System.out.println("El valor es correcto");
    resultado = 5;
}
else {
    System.out.println("El valor es incorrecto");
    resultado = 27;
}
```

Ejemplos condiciones

Por ejemplo:

```
int x = 3;
if(x == 3) {
    System.out.print("x es 3");
    x = 2;
}
else {
    System.out.print("x no es 3");
    x = 1;
}
System.out.print(x);
```

SALIDA

x es 32

```
int x = 5;
if(x == 3) {
    System.out.print("x es 3");
    x = 2;
}
else {
    System.out.print("x no es 3");
    x = 1;
}
System.out.print(x);
```

SALIDA

x no es 31

Condiciones anidadas

Podemos anidar tantas condiciones como necesitemos, aunque **no es recomendable tener muchos ifs anidados**, esto dificulta la lectura del código. Las llaves se pueden anidar dentro de las condiciones, de distintas formas, veamos algunos ejemplos:

A

```
int x = 0, y = 1;
if(x == 0) {
    x = 2;
    y = 2;
}
else {
    if(x == 0) {
        y = 3;
        x = 3;
    } else {
        y = 4;
        x = 4;
    }
}
```

x,y → 2

B

```
int x = 0, y = 1;
if(x == 1) {
    x = 2;
    y = 2;
}
else {
    if(x == 2) {
        y = 3;
        x = 3;
    } else {
        y = 4;
        x = 4;
    }
}
```

x,y → 4

C

```
int x = 2, y = 1;
if(x == 0) {
    x = 2;
    y = 2;
}
else {
    if(x == 2) {
        y = 3;
        x = 3;
    } else {
        y = 4;
        x = 4;
    }
}
```

x,y → 3

D

```
int x = 0, y = 1;
if(x == 1) {
    x = 2;
    y = 2;
}
else {
    if(x == 0) {
        y = 3;
        x = 3;
    }
    y = 4;
    x = 4;
}
```

x,y → 4

Condiciones anidadas

Podemos anidar tantas condiciones como necesitemos, aunque no es recomendable tener muchos ifs anidados, esto dificulta la lectura del código. Las llaves se pueden anidar dentro de las condiciones, de distintas formas, veamos algunos ejemplos:

A

```
int x = 0, y = 1;
if(x == 0)
    x = 2;
else
    if(x == 0)
        y = 3;
    else
        y = 4;
        x = 4;
```

$x \rightarrow 4, y \rightarrow 1$

B

```
int x = 0, y = 1;
if(x == 1)
    x = 2;
else {
    if(x == 2) {
        y = 3;
        x = 3;
    } else
        y = 4;
    x = 4;
}
```

$x, y \rightarrow 4$

C

```
int x = 2, y = 1;
if(x == 0)
    x = 2;
    y = 2;
else{
    if(x == 2) {
        y = 3;
        x = 3;
    } else {
        y = 4;
        x = 4;
    }
}
```

error

D

```
int x = 0, y = 1;
if(x == 1) {
    x = 2;
    y = 2;
}
x = 3;
else{
    if(x == 0) {
        y = 3;
        x = 3;
    }
    y = 4;
    x = 4;
}
```

error

Uso de las llaves en condiciones

Las llaves delimitan el trozo de código que se ejecutará según el resultado de la condición. Las llaves se pueden anidar dentro de las condiciones. **Podemos prescindir de las llaves si dentro del bloque if/else tenemos una única sentencia:**

```
if (a == 1) {  
    System.out.println("El valor es 1");  
} else {  
    if (a == 2) {  
        System.out.println("El valor es 2");  
    } else {  
        if (a == 3) {  
            System.out.println("El valor es 3");  
        } else {  
            System.out.println("El valor es mayor que 3");  
        }  
    }  
}
```

Uso de las llaves en condiciones

Las llaves delimitan el trozo de código que se ejecutará según el resultado de la condición. Las llaves se pueden anidar dentro de las condiciones. Podemos prescindir de las llaves si dentro del bloque if/else tenemos una única sentencia:

```
if (a == 1)
    System.out.println("El valor es 1");
else {
    if (a == 2) {
        System.out.println("El valor es 2");
    } else {
        if (a == 3) {
            System.out.println("El valor es 3");
        } else {
            System.out.println("El valor es mayor que 3");
        }
    }
}
```

Uso de las llaves en condiciones

Las llaves delimitan el trozo de código que se ejecutará según el resultado de la condición. Las llaves se pueden anidar dentro de las condiciones. Podemos prescindir de las llaves si dentro del bloque if/else tenemos una única sentencia:

```
if (a == 1)
    System.out.println("El valor es 1");
else {
    if (a == 2)
        System.out.println("El valor es 2");
    else {
        if (a == 3) {
            System.out.println("El valor es 3");
        } else {
            System.out.println("El valor es mayor que 3");
        }
    }
}
```

Uso de las llaves en condiciones

Las llaves delimitan el trozo de código que se ejecutará según el resultado de la condición. Las llaves se pueden anidar dentro de las condiciones. Podemos prescindir de las llaves si dentro del bloque if/else tenemos una única sentencia:

```
if (a == 1)
    System.out.println("El valor es 1");
else {
    if (a == 2)
        System.out.println("El valor es 2");
    else {
        if (a == 3)
            System.out.println("El valor es 3");
        else {
            System.out.println("El valor es mayor que 3");
        }
    }
}
```

Uso de las llaves en condiciones

Las llaves delimitan el trozo de código que se ejecutará según el resultado de la condición. Las llaves se pueden anidar dentro de las condiciones. Podemos prescindir de las llaves si dentro del bloque if/else tenemos una única sentencia:

```
if (a == 1)
    System.out.println("El valor es 1");
else if (a == 2)
    System.out.println("El valor es 2");
else if (a == 3)
    System.out.println("El valor es 3");
else
    System.out.println("El valor es mayor que 3");
```

Switch en JAVA

Condiciones (SWITCH)

Si queremos comprobar varias condiciones, podemos utilizar varios if/else encadenados, pero tenemos una forma alternativa de hacer esto en JAVA. Podemos elegir una opción entre los distintos valores posibles que tome una cierta expresión. Su formato es el siguiente:

```
switch (expresion) {  
    case valor1: sentencias1; break;  
    case valor2: sentencias2; break;  
    case valor3: sentencias3; break;  
    // ... Puede haber más valores  
    default: sentenciaD;  
}
```

Condiciones (SWITCH)

Es decir, después de la orden **switch** indicamos **entre paréntesis la expresión que queremos evaluar**. Después, tenemos distintos apartados, uno para cada valor que queramos comprobar; cada uno de estos apartados se precede con la palabra **case**, indica los pasos a dar si es ese valor el que tiene la variable (esta serie de pasos no será necesario indicarla entre llaves), y **termina** con **break**.

Un ejemplo sería:

```
switch (x*10) {  
    case 30: System.out.println("El valor de x es 3"); break;  
    case 40: System.out.println("El valor de x es 4");  
              System.out.println("Instrucción adicional"); break;  
    case 50: System.out.println("El valor de x es 5"); break;  
    case 60: System.out.println("El valor de x es 6"); break;  
}
```


Condiciones (SWITCH)

También podemos indicar qué queremos que ocurra en el caso de que el **valor de la expresión no sea ninguno** de los que hemos detallado, usando la palabra "**default**":

```
switch (expresion) {  
    case valor1: sentencias1; break;  
    case valor2: sentencias2; break;  
    case valor3: sentencias3; break;  
    // ... Puede haber más valores  
    default: sentencias; // Opcional: valor por defecto  
}
```

Por ejemplo así:

```
switch (x*10) {  
    case 30: System.out.println("El valor de x era 3"); break;  
    case 50: System.out.println("El valor de x era 5"); break;  
    case 60: System.out.println("El valor de x era 6"); break;  
    default: System.out.println("El valor de x no era 3, 5 ni 6"); break;  
}
```

Condiciones (SWITCH)

Podemos conseguir que ejecuten las mismas sentencias en varios casos, simplemente eliminando la orden "break" de algunos de ellos. Por ejemplo:

```
switch (x) {  
    case 1:  
    case 2:  
    case 3:  
        System.out.println("El valor de x estaba entre 1 y 3"); break;  
    case 4:  
    case 5:  
        System.out.println("El valor de x era 4 o 5"); break;  
    case 6:  
        System.out.println("El valor de x era 6");  
        x = 10;  
        System.out.println("Operaciones auxiliares completadas"); break;  
    default:  
        System.out.println("El valor de x no estaba entre 1 y 6");  
}
```

El operador ternario

El operador ternario

Existe una construcción adicional, que **permite comprobar si se cumple una condición y devolver un cierto valor según si dicha condición se cumple o no**. Es lo que se conoce como el "operador ternario" (?).

```
variable = condicion ? resultado_si_cierto : resultado_si_falso
```

Es decir, se indica la condición seguida por una interrogación, después el valor que hay que devolver si se cumple la condición, a continuación un símbolo de "dos puntos", y finalmente el resultado que hay que devolver si no se cumple la condición.

El operador ternario

Es frecuente emplearlo en asignaciones (aunque algunos autores desaconsejan su uso porque puede resultar menos legible que un "if"), como en este ejemplo:

```
x = a == 10 ? b*2 : a ;
```

En este caso, si "a" vale 10, la variable "x" tomará el valor de b*2, y en caso contrario, tomará el valor de "a". Esto también se podría haber escrito de la siguiente forma, más larga pero más legible:

```
if (a == 10)
    x = b*2;
else
    x = a;
```

El operador ternario

Podemos concatenar ternarias en ambos resultados, si lo hacemos en la parte del **true**:

```
int a = 10, b = 10, x;  
x = a == 10 ? b == 10 ? 100 : 1 : a ; //100
```

Podemos concatenar ternarias en ambos resultados, si lo hacemos en la parte del **false**:

```
int a = 1, b = 10, x;  
x = a == 10 ? b*2 : b == 10 ? 100 : a; //100
```

Ejemplos ternarias

Podemos concatenar todas las ternarias que necesitemos, veamos algunos ejemplos.

```
int a = 1, b = 2, c = 3;
boolean condicion = a == 1 ? true : false;
if (condicion) System.out.println("a vale 1");
else System.out.println("a no vale 1");

condicion = a == 3 ? true :
            b == 3 ? true :
            c == 3 ? true : false;
if (condicion) System.out.println("Alguna variable vale 3");
else System.out.println("Ninguna variable vale 3");

a = a != 0 ? 2 : 3; //a vale 2
b = a == c ? 2 : 1; //b vale 1
```

Bucles

Estructuras de control repetitivas en JAVA

Repetitivas: bloque de código que puede ejecutarse más de una vez.

while (condición) {

```
instrucción1;  
instrucción2;  
.....  
instrucciónN;  
}
```

```
instrucción4;
```

- ✓ Se ejecutará mientras la condición sea cierta.
- ✓ Es posible que nunca se ejecuten las instrucciones.

do {

```
instrucción1;  
instrucción2;  
.....  
instrucciónN;
```

```
}  
while (condición);
```

```
instrucción4;
```

- ✓ Se ejecutará mientras se cumpla la condición.
- ✓ Las instrucciones se ejecutan como mínimo una vez.

for (Vi; Vf; Vc) {

```
instrucción1;  
instrucción2;  
.....  
instrucciónN;
```

```
}  
instrucción4;
```

- ✓ Se ejecutará mientras se cumpla la condición.
- ✓ Las instrucciones se ejecutan un número determinado de veces.
- ✓ Se indica el valor inicial(Vi), valor final(Vf) y el incremento(Vc).

Bucle while

Con frecuencia tendremos que hacer que **una parte de nuestro programa se repita** (algo a lo que con frecuencia llamaremos "bucle"). Este **trozo** de programa **se puede repetir mientras se cumpla una condición** o bien un cierto número prefijado de veces.

BUCLE WHILE

Java incorpora **varias formas** de conseguirlo. La primera que veremos es la orden **"while"**, que hace que una parte del programa se repita mientras se cumpla una cierta condición. Su **formato** será:

```
while (condición) {  
    sentencia1;  
    sentencia2;  
    ...  
    sentenciaN;  
}
```

Bucle do-while

Existe una **variante** de este tipo de bucle. Es el conjunto **do-while**, cuyo formato es:

BUCLE DO-WHILE

En este caso, la condición se comprueba al final, lo que quiere decir que **las sentencias intermedias se realizarán al menos una vez**, cosa que no ocurría en la construcción anterior, porque con **"while"**, **si la condición era falsa**, las **sentencias** de dentro del "while" **no llegaban a ejecutarse nunca**.

```
do {  
    sentencia1;  
    sentencia2;  
    ...  
    sentenciaN;  
} while (condición);
```

Ejemplos while y do-while

Ejemplos **bucle while** y **do-while**, ambos bucles **hacen lo mismo**:

```
int a = 0, b = 1;
while (a < 4) {
    b+=a;
    a++;
} //a → 4, b → 7
```

```
int a = 0, b = 1;
do {
    b+=a;
    a++;
} while (a < 4);
//a → 4, b → 7
```

Ejemplo adicional **modificando el orden de las instrucciones** de cuerpo del bucle:

```
int a = 0, b = 1;
while (a < 4) {
    a++;
    b+=a;
} //a → 4, b → 11
```

```
int a = 0, b = 1;
do {
    a++;
    b+=a;
} while (a < 4);
//a → 4, b → 11
```

Ejemplos while y do-while

El bucle **do-while** se utiliza cuando tengamos que hacer como mínimo una ‘pasada’ por el cuerpo del

```
Scanner teclado = new Scanner(System.in);
int num = 0;
System.out.println("Introduce un número distinto de cero para seguir en el bucle");
num = teclado.nextInt(); //se queda a la espera
while (num != 0) { //si introduzco cualquier número diferente a cero entro
    System.out.println("Introduce un número distinto de cero para seguir en el bucle");
    num = teclado.nextInt();
    ... //aquí van más instrucciones
}
```

WHILE

```
Scanner teclado = new Scanner(System.in);
int num;
do { //con un do while nos evitamos escribir dos veces el sout como arriba
    System.out.println("Introduce un número distinto de cero para seguir en el bucle");//1 vez
    num = teclado.nextInt();
    ... //aquí van más instrucciones
} while (num != 0);
```

DO-WHILE

Bucle for

Una tercera forma de conseguir que parte de nuestro programa se repita es la orden "for". La emplearemos sobre todo para conseguir un número concreto de repeticiones.

```
for (valor_inicial ; condicion_continuacion ; incremento) {  
    sentencia1;  
    sentencia2;  
    ...  
    sentenciaN;  
}
```

BUCLE FOR

Debemos indicar entre parentesis, y separadas por puntos y coma, tres ordenes:

- La **primera** orden dará el valor inicial a una variable que sirve de control.
- La **segunda** orden será la condición del bucle, mientras sea cierta, permaneceremos dentro del bucle.
- La **tercera** orden será la que se encargue de modificar la variable de control para que en algún momento la condición se deje de cumplir y el bucle termine

Bucle for

Esto se verá mejor con varios ejemplos. Podríamos repetir varias veces un bloque de instrucciones haciendo:

```
for (int i = 0; i < 10; i++) { ... }
```

```
for (int i = 1; i <= 10; i++) { ... }
```

```
for (int i = 0; i <= 10; i++) { ... }
```

```
for (int i = 1; i < 10; i++) { ... }
```

Inicialmente 'i' tiene como valores 0 y 1. Hay que **repetir mientras se cumpla la condición**, y en cada pasada por el bucle hay que aumentar el valor de 'i' en una unidad.

Bucle for

Esto se verá mejor con varios ejemplos. Podríamos repetir varias veces un bloque de instrucciones haciendo:

```
for (int i = 0; i < 10; i++) { ... } //10 pasadas
```

```
for (int i = 1; i <= 10; i++) { ... } //10 pasadas
```

```
for (int i = 0; i <= 10; i++) { ... } //11 pasadas
```

```
for (int i = 1; i < 10; i++) { ... } //9 pasadas
```

Inicialmente 'i' tiene como valores 0 y 1. Hay que **repetir mientras se cumpla la condición**, y en cada pasada por el bucle hay que aumentar el valor de 'i' en una unidad.

Orden de ejecución de instrucciones

Es muy importante conocer el orden de ejecución de las instrucciones en un bucle for.

```
for (int j = 10 ; j > 0 ; j -= 2 ) {  
    j+=1;  
    System.out.print(j);  
}  
System.out.println(j);
```

1. Declaramos un contador y lo inicializamos (se ejecuta una única vez).
2. Se comprueba la condición, si se cumple, pasamos al **paso 3**, de lo contrario el bucle termina.
3. Se ejecuta el cuerpo del bucle.
4. Modificamos el contador.
5. Volvemos al **paso 2**.

Orden de ejecución de instrucciones

Es muy importante conocer el orden de ejecución de las instrucciones en un bucle for.

```
for (int j = 10 ; j > 0 ; j -= 2 ) {  
    j+=1;  
    System.out.print(j); //111098765432  
}  
System.out.println(j); //error
```

1. Declaramos un contador y lo inicializamos (se ejecuta una única vez). **//j=10**
2. Se comprueba la condición, si se cumple, pasamos al **paso 3**, de lo contrario el bucle termina. **//j>10**
3. Se ejecuta el cuerpo del bucle. **// j +=1= 11 -- imprimimos j**
4. Modificamos el contador. **//j-=2 j=9**
5. Volvemos al **paso 2**.

Bucle for

A continuación se muestran un ejemplo de bucle FOR con su salida correspondiente:

```
for(j = 10 ; j > 0 ; j -= 2) {  
    System.out.println(j);  
}
```



10
8
6
4
2

```
for(j = 10 ; j >= 0 ; j -= 2) {  
    System.out.println(j);  
}
```



10
8
6
4
2
0

Bucle for

Si el **bucle** tiene una **única instrucción** **no** es **necesario** incluir las **llaves**.

```
for (int j = 1; j < 10; j += 2)  
    System.out.println(j);
```



1
3
5
7
9

```
for (int j = 0; j <= 10; j += 2)  
    System.out.println(j);
```



0
2
4
6
8
10

Bucle for

Se puede observar una **equivalencia** casi inmediata **entre** la orden **"for"** y la orden **"while"**. Así, el ejemplo anterior se podría reescribir empleando "while", de esta manera:

```
for (int j = 10; j > 0; j -= 2)
    System.out.print(j);
```



```
j = 10;
while (j > 0) {
    System.out.print(j);
    j -= 2;
}
```

Precaución con los bucles: Casi siempre, nos interesará que una parte de nuestro programa se repita varias veces (o muchas veces), pero no indefinidamente. **Si planteamos mal la condición de salida**, nuestro programa se puede quedar "colgado", repitiendo de forma indefinida las instrucciones del cuerpo del bucle.

Ejemplos Bucles

Es muy importante tener claro el orden de ejecución de las instrucciones en un **bucle for**:

```
for(j = 0 ; j < 3 ; j++)  
    System.out.println(j);
```



Orden	Instrucción	j
1	j = 0	0
2	j < 3	0
3	println	0
4	j++	1
5	j < 3	1
6	println	1
7	j++	2
8	j < 3	2
9	println	2
10	j++	3
11	j < 3	3
12	FIN	

Ejemplos Bucles

Es muy importante tener claro el orden de ejecución de las instrucciones en un **bucle for**:

```
for(j = 3 ; j >= 0 ; j--)  
    System.out.println(j);
```



Orden	Instrucción	j
1	j = 3	3
2	j >= 0	3
3	println	3
4	j--	2
5	j >= 0	2
6	println	2
7	j--	1
8	j >= 0	1
9	println	1
10	j--	0
11	j >= 0	0
12	println	0
13	j--	-1
14	j >= 0	-1
12	FIN	

Ejemplos Bucles

Es muy importante tener claro el orden de ejecución de las instrucciones en un **bucle for**:

```
for(j = 10 ; j > 4 ; j -= 2)  
    System.out.println(j);
```



Orden	Instrucción	j
1	j = 10	10
2	j > 4	10
3	println	10
4	j -= 2	8
5	j > 4	8
6	println	8
7	j -= 2	6
8	j > 4	6
9	println	6
16	j -= 2	4
17	j > 4	4
18	FIN	

Sintaxis Bucle for

La cláusula de inicio y la cláusula de modificación pueden estar compuestas por varias expresiones separadas mediante el operador coma (,).

```
for(int j = 1, i = 1; j < 5; j++) {  
    System.out.print(j+i); //2345  
}
```

```
for(int j = 1; j < 10; j++, j += 2) {  
    System.out.print(j); //147  
}
```

```
for(int j = 1, i = 5; j < i; j+=2, i++) {  
    System.out.print(j); //1357  
}
```

Break

Se puede modificar el comportamiento de los bucles con las órdenes "**break**" y "**continue**". La sentencia "**break**" hace que se salten las instrucciones del bucle que quedan por realizar, y se salga del bucle inmediatamente.

```
System.out.println("Empezamos...");  
for ( i = 1 ; i <= 10 ; i++ ){  
    System.out.println("Vuelta: "+i);  
    if (i == 8)  
        break;  
    System.out.println("Terminada vuelta: "+i);  
}  
System.out.println("Terminado");
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, ni se darían las pasadas de i=9 e i=10, porque ya se ha abandonado el bucle.

Continue

La sentencia "**continue**" hace que se salten las instrucciones del bucle que quedan por realizar, pero no se sale del bucle sino que se pasa a la siguiente iteración (la siguiente "vuelta" o "pasada").

```
System.out.println("Empezamos...");  
for (i = 1 ; i <= 10 ; i++){  
    System.out.println("Vuelta: "+i);  
    if (i == 8)  
        continue;  
    System.out.println("Terminada vuelta: "+i);  
}  
System.out.println("Terminado");
```

En este caso, no se mostraría el texto "Terminada esta vuelta" para la pasada con i=8, pero sí se darían la pasada de i=9 y la de i=10.

Break y Continue etiquetados

Break y continue pueden “saltar” a una etiqueta con el operador dos puntos (:). Sin embargo, no se recomienda utilizarlos por su parecido al infame goto de C++. Este tipo de estructuras no son adecuadas para seguir fácilmente el flujo del programa.

```
System.out.println("Empezamos...");
for (i = 1 ; i <= 10 ; i++){
    System.out.println("Vuelta: "+i);
    salir:
    if (i == 8){
        i++
        break salir;
    }
    System.out.println("Terminada vuelta: "+i);
}
System.out.println("Terminado");
```

Sintaxis Bucle for

Podemos prescindir de cualquiera de las instrucciones de la cabecera del bucle for. Sin embargo, si no ponemos condición, el bucle no terminará nunca. Aunque podemos incluir dentro del cuerpo del bucle un break, no es recomendable hacerlo.

```
for(int j = 1; j < 5; j++) { ... }
```

```
for( ; j < 5; j++) { ... }
```

```
for( ; j < 5; ) { ... }
```

```
for( ; ; ) { ... }
```

```
int n = 0;
```

```
for(; ;) {
```

```
    if(n == 5) break;
```

```
    System.out.print(n);
```

```
    n++;
```

```
}
```

```
System.out.println("Fin");
```

Bucles anidados

Ejemplos Bucle anidado

En programación es muy habitual encontrar dos bucles for anidados:

```
int a=4, b=3;

for(int j = 0; j < a; j++) {
    for(int i = 0; i < b; i++) {
        System.out.print(i);
    }
}
```

```
int a=3, b=2;

for(int j = 1; j <= a; j++) {
    for(int i = 1; i <= b; i++) {
        System.out.print(j);
    }
}
```

Ejemplos Bucle anidado

En programación es muy habitual encontrar dos bucles for anidados:

```
int a=4, b=3;

for(int j = 0; j < a; j++) {
    for(int i = 0; i < b; i++) {
        System.out.print(i);
    }
} //012012012012
```

```
int a=3, b=2;

for(int j = 1; j <= a; j++) {
    for(int i = 1; i <= b; i++) {
        System.out.print(j);
    }
} //112233
```


Ejemplos Bucles anidados

Al tener 2 bucles anidados es muy importante saber la cantidad de iteraciones(vueltas) que hace tanto el bucle interno, como el externo:

```
int cantidad = 0;
for(int j = 0; j < 100; j++) {
    for(int i = 0; i < 10; i++) {
        cantidad++;
    }
}
```

```
int cantidad = 0;
for(int j = 0; j < 100; j++) {
    for(int i = 0; i <= 10; i++) {
        cantidad++;
    }
}
```

```
int cantidad = 0;
for(int j = 1; j < 100; j++) {
    for(int i = 1; i < 10; i++) {
        cantidad++;
    }
}
```

```
int cantidad = 0;
for(int j = 1; j <= 100; j++) {
    for(int i = 1; i < 10; i++) {
        cantidad++;
    }
}
```

Ejemplos Bucles anidados

Al tener 2 bucles anidados es muy importante saber la cantidad de iteraciones(vueltas) que hace tanto el bucle interno, como el externo:

```
int cantidad = 0;
for(int j = 0; j < 100; j++) {
    for(int i = 0; i < 10; i++) {
        cantidad++;
    }
} // 1000 (100*10)
```

```
int cantidad = 0;
for(int j = 0; j < 100; j++) {
    for(int i = 0; i <= 10; i++) {
        cantidad++;
    }
} // 1100 (100*11)
```

```
int cantidad = 0;
for(int j = 1; j < 100; j++) {
    for(int i = 1; i < 10; i++) {
        cantidad++;
    }
} // 891 (99*9)
```

```
int cantidad = 0;
for(int j = 1; j <= 100; j++) {
    for(int i = 1; i < 10; i++) {
        cantidad++;
    }
} // 900 (100*9)
```

Ejemplos Bucles

A continuación se muestra un ejemplo con dos bucles anidados. Como se puede observar, el primer caso prescinde de las llaves para el bucle externo. **Es completamente válido**, pero debemos tener en cuenta algo, el bucle externo tiene una única instrucción (el bucle interno);

```
int a=5, b=2, c=3;

for(int j=1; j<=a; j++)
    for(int i=1; i<=b; i++) {
        c=a++;
        b--;
    }
```

```
int a=5, b=2, c=3;

for(int j=1; j<=a; j++) {
    for(int i=1; i<=b; i++) {
        c=a++;
        b--;
    }
}
```

Ejemplos Bucles

A continuación se muestran un ejemplo con dos bucles anidados:

```
int a=5, b=2, c=3;

for(int j=1; j<=a; j++){
    for(int i=1; i<=b; i++) {
        c=a++;
        b--;
    }
}
```



no entra, $i > b$

si entra, $i = b$

a	b	c	i	j
5	2	3		
6	1	5	1	1
6	1	5	2	1
6	1	5	1	2
7	0	6	1	2
7	0	6	1	3
7	0	6	1	4
7	0	6	1	5
7	0	6	1	6
7	0	6	1	7
7	0	6		

Llaves y ámbito de las variables

Llaves y ámbito de las variables

En JAVA las **llaves** se utilizan para **delimitar bloques** de código. **Normalmente** se utilizan para **delimitar el código** que se ejecuta cuando se **cumple una cierta condición**, por ejemplo en condicionales y bucles. Sin embargo, no son obligatorias si el bloque de código contiene una única sentencia. Por otra parte, hay otras **estructuras** que **obligatoriamente siempre van entre llaves**:

- El código de una clase.
- Los métodos o funciones.

Las **llaves se pueden incluir en cualquier parte** del código, sin que sea necesario que estén asociadas a bucles, condiciones, métodos o clases. Además, **podemos anidar llaves** dentro de otras llaves tanto veces como lo necesitemos. Pero al utilizar llaves, sucede **algo muy importante**, al **declarar una variable dentro de unas llaves, dicha variable sólo existirá dentro de esas llaves**, si intentamos acceder desde fuera, se producirá un error de compilación.

Llaves y ámbito de las variables

```
{ int a = 1, b = 2, c = 3; }  
a = 3; //error
```

```
{ int a = 1, b = 2, c = 3;  
a = 3; } //ok
```

```
{ int a = 1, b = 2, c = 3;  
  { int d = 4; }  
  d = 5; //error. ámbito dif  
}
```

```
{ int a = 1, b = 2, c = 3;  
  int d = 4;  
  d = 5; //ok  
}
```

```
for(int j = 1; j <= a; j++) {  
    System.out.print(j);  
}  
j = 3; //error j ámbito for
```

```
int j = 1  
for( ; j <= a; j++)  
    System.out.print(j);  
j = 3; //ok
```

Llaves y ámbito de las variables

En JAVA podemos declarar variables con el mismo identificador si están en ámbitos (scope) distintos, el ámbito lo delimita un bloque de código que está entre llaves.

Sin embargo, se debe tener en cuenta que si anidamos llaves dentro de otras llaves, el bloque interno tiene el mismo ámbito que el bloque de llaves externo.

```
for(int j = 1; j < 5; j++) {  
    System.out.print(j);  
}  
for(int j = 1; j < 10; j++) {  
    System.out.print(j);  
}  
int j = 3; //ok
```


Llaves y ámbito de las variables

En JAVA podemos declarar variables con el mismo identificador si están en ámbitos (scope) distintos, el ámbito lo delimita un bloque de código que está entre llaves.

Sin embargo, se debe tener en cuenta que si anidamos llaves dentro de otras llaves, el bloque interno tiene el mismo ámbito que el bloque de llaves externo.

```
for(int j = 1; j < 5; j++) {  
    System.out.print(j);  
} SCOPE-1  
for(int j = 1; j < 10; j++) {  
    System.out.print(j);  
} SCOPE-2  
int j = 3; //ok SCOPE-3
```

```
for(int j = 1; j < 5; j++) {  
    System.out.print(j); SCOPE-1  
    for(int i = 1; i < 10; i++) {  
        System.out.print(i);  
    }  
} SCOPE-2  
int j = 3; //ok SCOPE-3
```

Llaves e indentado del código

En programación existen distintos estilos para indentar el código incluido dentro de las llaves:

- **Kerninghan y Ritchie(K&R)**: es el más utilizado, utiliza menos líneas para código, también se le denomina “**el estilo verdadero de llaves**”. Es el estilo utilizado en el núcleo de Unix.
- **Allman**: las llaves siempre van solas, no tiene código ni a la izquierda, ni a la derecha. Es **menos utilizado por utilizar más líneas** para representar el mismo código que K&R.

Indentado automático VSCODE

- Windows (Alt+Shift+F)
- Linux (Ctrl+Shift+I)

```
for(int i=1; i<=b; i++)  
{  
    c=a++;  
    b--;  
}
```

```
for(int i=1; i<=b; i++) {  
    c=a++;  
    b--;  
}
```

Conversión de tipos: casting y parse

Conversión de tipos

Internamente JAVA puede hacer conversiones de tipos en determinadas ocasiones, siempre que dicha conversión sea posible, por ejemplo:

- La función `Math.pow(a,b)`, recibe 2 parámetros de tipo `double`, podemos pasarle dos enteros sin problemas, JAVA hará esta conversión de forma automática.

En cambio, si intentamos almacenar en un entero, un valor de tipo `double` se producirá un error si no hacemos una conversión explícita (casting). Veamos un ejemplo:

```
int a = Math.random(); //error
```

En estos casos podemos indicar a JAVA que haga una conversión de tipos (casting), siempre y cuando los tipos sean compatibles.

```
int a = (int)Math.random(); //0
```

Parse

- También tenemos la opción de **convertir cadenas de texto(String) a valores numéricos**. Esto nos permite trabajar con dicho valor como con cualquier otro dato primitivo (como son int, float, etc.)
- Para realizar esta conversión es **necesario usar "clases envoltura"**, las cuales nos permiten tratar tipos primitivos como objetos.
- Normalmente, la envoltura posee el mismo nombre que el tipo de dato primitivo, aunque con la primera letra en mayúscula.

```
String cadena = "12345";  
int myInt = Integer.parseInt(cadena);
```

Parse

A continuación tenemos una tabla con los 'parse' de las distintas clases envoltura de JAVA, podemos usarlas para parsear variables de tipo **String**.

Tipo	Clase de envoltura	Método
byte	Byte	Byte.parseByte(aString)
short	Short	Short.parseShort(aString)
int	Integer	Integer.parseInt(aString)
long	Long	Long.parseLong(aString)
float	Float	Float.parseFloat(aString)
double	Double	Double.parseDouble(aString)
boolean	Boolean	Boolean.parseBoolean(aString)

Parse

A continuación se muestran ejemplos parseando un double y un boolean.

```
String real = "12.35";  
double d = Double.parseDouble(real);  
System.out.println(d*2); //27.7  
real = "true";  
boolean condicion = Boolean.parseBoolean(real);  
if(condicion) System.out.println(condicion); //true
```

Para convertir un char a entero utilizamos un casting, y para convertir char a String utilizamos Character.toString():

```
char letra = 'a';  
int entero = (int)letra; //97  
String frase = Character.toString(letra); //a
```

Preguntas

