CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN

UNIDAD 1

INTRODUCCIÓN A LA PROGRAMACIÓN

DIEGO VALERO ARÉVALO
BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
Mª CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

U1 - **INTRODUCCIÓN A LA PROGRAMACIÓN**
ÍNDICE

1.1 - LO BÁSICO

¡Bienvenido al asombroso mundo de la **programación**! En este temario vamos a ver qué es **programar**, qué son los **algoritmos** e iremos yendo paso a paso viendo las funciones básicas de **JAVA** mientras creamos diferentes programas. ¡Vamos allá!

>> 1.1.1- ¿QUÉ ES LA PROGRAMACIÓN?

Antes de llegar a una conclusión, veamos qué compone la **programación** a grandes rasgos:

PROGRAMA	DATO	ALGORITMO
Es una **secuencia de instrucciones** con **gramática** y **sintaxis** que un ordenador puede **interpretar y ejecutar**.	Es una **representación simbólica de un atributo** o **variable cuantitativa** o **cualitativa**. Representa la **información que el programador manipula** cuando **crea una solución** a un problema.	Es un **conjunto ordenado y finito de operaciones** que permite **hallar la solución de un problema**.

Un ordenador es **muy tonto**. Sí, aunque parezca que es la máquina más inteligente del mundo, literalmente **no sabe hacer nada si nosotros no se lo decimos**. Por eso, para que pueda funcionar, necesita unos **datos de entrada** que se **ejecutarán** con el objetivo de dar **datos de salida**, o mejor dicho, la **solución a un problema**.

Este problema no lo soluciona el ordenador él solo, sino que por detrás, un humano (conocido como **programador**) ha conseguido **crear un programa que soluciona ese problema**, para que al darle unos datos a la máquina, ésta sepa cómo utilizarlos en el algoritmo.

Un **dato** por sí mismo **no constituye información**, es el **procesamiento de los datos** lo que nos proporciona **información**.

>> 1.1.2- ALGORITMOS

Como ya hemos visto, los **algoritmos** son un **conjunto ordenado y finito de operaciones** que permite **hallar la solución de un problema**. Un algoritmo cumple las siguientes características:

- Tiene un **número finito de pasos**.
- **Acaba** en un **tiempo finito** (si no acabase nunca, no se resolvería el problema).
- Todas las **operaciones** deben estar **definidas de forma precisa**.
- Puede tener **varios datos de entrada** y como **mínimo uno de salida**.

Un ejemplo típico de algoritmo puede ser una **receta de cocina**:

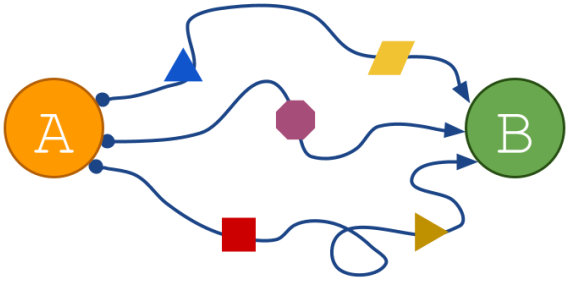
PREPARAR UN SANDWICH MIXTO A LA PLANCHA
DATOS DE ENTRADA
Pan, jamón york y queso

INSTRUCCIONES
- Sacar dos rebanadas de pan de la bolsa. - Poner jamón york sobre una de las rebanadas. - Poner queso sobre la rebanada. - Tapar con la otra rebanada. - Encender y calentar plancha. - Poner el sándwich sobre la plancha durante 30 segundos. - Girar el sandwich y dejar otros 30 segundos. - Apagar plancha.
DATOS DE SALIDA
Sándwich mixto

Debemos tener en cuenta que **algunas instrucciones pueden alterar su orden**, pero otras **no se pueden ejecutar si no se han ejecutado otras antes**. Por ejemplo, **no se puede apagar la plancha** si **no se ha encendido**.

>> 1.1.3- PARADIGMAS DE LA PROGRAMACIÓN

Los paradigmas de la programación son **cada uno de los caminos que podemos tomar** para llegar a la **solución** de un problema, porque **no existe una única manera universal para ello**, sino muchas.



En el mundo de la programación, los paradigmas son **estilos documentados para programar**, cada estilo es **distinto**, y cada uno tiene sus **ventajas** e **inconvenientes**, pero **con todos se obtiene el mismo resultado final**.

PROGRAMA IMPERATIVO	PROGRAMA FUNCIONAL
Es aquel que nos dice **qué hacer paso por paso**. En él se describe de forma **secuencial** todo lo que hará el programa.	Se divide el problema en **pedazos o módulos** (**función**) donde cada uno **hace una sola cosa**.
PROGRAMA LÓGICO	**PARADIGMA ORIENTADO A OBJETOS**
Se basa en el **pensamiento lógico y natural**. De esta manera se pueden **expresar de manera formal problemas complejos**, elaborando **premisas** y luego aplicando **hipótesis**, **axiomas** y **teoremas** para la resolución. La programación lógica resulta óptima en aplicaciones de **inteligencia artificial**.	Aquel donde la funcionalidad se aplica a **elementos** (**objetos**) que tienen **características** y **funciones**. Esta forma de programar hace más fácil manejar y mantener un sistema, si necesitamos una nueva funcionalidad, podemos agregar **un nuevo objeto** o **añadir datos y funcionalidades** a los objetos **que ya existen**.

>> 1.1.4- LENGUAJES DE PROGRAMACIÓN

Un **lenguaje de programación** es **aquello que entiende el ordenador** para poder **componer acciones y manejar datos y algoritmos** para, de esa forma, **crear programas** que controlen el comportamiento físico y lógico de una máquina. Es la **manera** en la que **se comunican el programador y la máquina**.

Tenemos dos tipos de lenguaje: de **bajo nivel** y de **alto nivel**.

BAJO NIVEL

Son aquellos lenguajes que **se orientan totalmente a la máquina**.

LENGUAJE MÁQUINA

Es el **más primitivo de los lenguajes** que se compone de una colección de **dígitos binarios** o **bits** (`0` y `1`) que la computadora **lee e interpreta**. Son los únicos idiomas que **entienden** las máquinas.

LENGUAJE ENSAMBLADOR

Este es el primer intento de **sustitución del lenguaje de máquina por uno más cercano al utilizado por los humanos**. Un programa escrito en éste lenguaje es **almacenado como texto** (tal como programas de **alto nivel**) y consiste en una serie de instrucciones que corresponden al **flujo de órdenes ejecutables por un microprocesador**. Sin embargo, dichas máquinas **no comprenden el lenguaje ensamblador**, por lo que **se debe convertir a lenguaje máquina** mediante un programa llamado **Ensamblador**.

ALTO NIVEL

Tienen como objetivo **facilitar el trabajo del programador**, ya que utilizan el lenguaje humano para crear **instrucciones más fáciles de entender**. Además, el lenguaje de alto nivel permite **escribir códigos en diferentes idiomas** y luego, para ser ejecutados, se traduce al lenguaje máquina mediante **traductores** o **compiladores**.

TRADUCTOR

Traducen programas escritos en **un lenguaje de programación** al **lenguaje máquina**, y a medida que va siendo traducida, **se ejecuta**.

COMPILADOR

Permite **traducir todo un programa de una sola vez**, haciendo **una ejecución más rápida** y **puede almacenarse para usarse luego** sin volver a hacer la traducción.

Además, el **lenguaje de alto nivel** se divide en **dos tipos**:

LENGUAJE INTERPRETADO

Es aquel en el cual sus **instrucciones**, o más bien el **código fuente**, es traducido **por el intérprete** a un **lenguaje entendible para la máquina paso a paso**, **instrucción por instrucción**. El proceso se **repite** cada vez **que se ejecuta el programa** del código en cuestión.

Ejemplos: `Ruby`, `Python`, `PHP`.

LENGUAJE COMPILADO

Es aquel cuyo **código fuente** es **traducido por un compilador** a un **archivo ejecutable entendible según la plataforma de la máquina**. Con ese archivo se puede **ejecutar el programa cuantas veces sea necesario sin tener que repetir el proceso** por lo que el tiempo de espera entre ejecución y ejecución es ínfimo.

Ejemplos: `JAVA`, `C++`, `Fortran`.

>> 1.1.5- DISEÑO DE ALGORITMOS: PSEUDOCÓDIGO

Antes de tirarnos a programar, debemos **estudiar el problema en papel** y generar el **algoritmo** para hallar la solución. Se utilizan una **mezcla de frases en lenguaje común**, y **palabras claves** que indican el **inicio** y el **fin** del algoritmo y las **instrucciones** específicas a realizar. El ejemplo del algoritmo con la receta del sándwich mixto es también un ejemplo de pseudocódigo. Vamos a ver otro más:

CALCULAR EL ÁREA DE UN RECTÁNGULO
DATOS DE ENTRADA
Medidas de base (b) y altura (h)
INSTRUCCIONES
- Inicio - Leer b - Leer h - a=b*h - Escribir a - Fin
DATOS DE SALIDA
Área del rectángulo

>> 1.1.6- CALIDAD DE PROGRAMAS

Todo software contiene **errores**, conocidos como **bugs**. Incluso en una simple calculadora es probable la existencia de estos, y aunque **no podamos controlarlos todos**, lo importante es **minimizar** la cantidad de los mismos, y más en programas complejos.

Para **minimizar bugs**, hay que partir de un **buen diseño** e ir **codificando y probando poco a poco** para enfrentarse a errores escalonadamente. No es nada recomendable crear todo el programa y probarlo después.

Otro aspecto muy importante es **documentar** y **comentar el código correctamente** en casi todos los aspectos necesarios.

Echa un vistazo a esta página para leer varias **historias interesantes** sobre cómo diferentes **bugs** de software a lo largo de la historia tuvieron consecuencias importantes:

LOS GRANDES ERRORES DE LA HISTORIA DEL SOFTWARE INFORMÁTICO - DEL PRIMER BUG A HOY >

1.2 - CONOCIENDO JAVA

Vamos a adentrarnos en el lenguaje que vamos a estar usando para programar los próximos meses: **JAVA**.

>> 1.2.1- LAS BASES DE JAVA

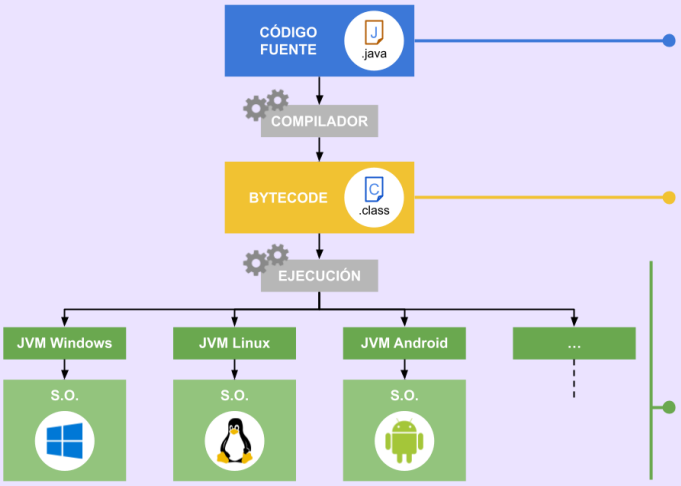
¿Qué hace a JAVA ser **JAVA**? Estas son sus bases:

Sigue el **paradigma de la programación** de la **orientación a objetos**, aunque a la hora de programar también se usa parte de **funcional**.

Es un **lenguaje de alto nivel**, lo que significa que usa **palabras comprensibles** por el ser humano.

Es **compilado**, por lo que antes de poder ponerlo en marcha, el código **se debe traducir sin errores**.

Es **multiplataforma**, haciendo que un mismo código se pueda ejecutar en **cualquier máquina y sistema operativo**. Esto es posible gracias a la **Máquina Virtual de Java (JVM)**, que es un **entorno de ejecución para aplicaciones Java**, y **adapta** los programas Java compilados a las características del sistema operativo donde se va a ejecutar.



Comenzamos con nuestro **código fuente** en un archivo `.java`.

Al **compilarlo** se generan uno o varios archivos denominados **bytecodes** cuya extensión es `.class`. Estos son **independientes de la arquitectura del S.O.**.

Los bytecodes serán los que ejecute la **JVM**, donde cada S.O. tiene su **propia implementación de esta**.

>> 1.2.2- HERRAMIENTAS DE JAVA E INSTALACIÓN

· 1.2.2.1- JRE Y JDK

Al utilizar JAVA, debemos tener en cuenta las **dos herramientas principales** que usa:

JRE	JDK
Es el **Java Runtime Environment** o **Entorno de Ejecución de Java**. Contiene a la **JVM** y **otras herramientas** que permiten la **ejecución** de las aplicaciones Java. JRE **no posee compiladores ni herramientas para desarrollar las aplicaciones Java**, solo posee las herramientas para **ejecutarlas**.	Es el **Java Development Kit** o **Herramientas de Desarrollo de Java** y sirve para **construir programas usando el lenguaje de programación JAVA**. Trae herramientas útiles como el **compilador** (**javac**), el **desensamblador de binarios** (**javap**) y el **depurador** (**debugger**), entre otras. Una instalación de JDK **ya contiene un JRE dentro de la misma**.

Antes de ponernos a trabajar con JAVA, debemos **instalar en nuestra máquina el JDK**. Puedes descargarlo desde la página de Oracle:

ORACLE DOWNLOADS: JDK >

Se recomienda instalar la versión revisada **anterior a la más reciente**, ya que las nuevas versiones siempre están en desarrollo y pueden tener errores. (Por ejemplo, si la última versión del JDK es la **20**, es más recomendable usar la **17**).

· 1.2.2.2- VARIABLES DE ENTORNO

Cuando tengamos instalado el JDK, antes de poder **utilizar sus herramientas** para **compilar** y **ejecutar**, es necesario **configurar las variables de entorno**. En Windows 10, lo haremos de la siguiente forma:

Panel de control > Sistema y Seguridad > Sistema > Configuración avanzada del sistema > Variables de entorno

Deberemos **crear una nueva variable**. Para ello:

1. Damos como nombre `JAVA_HOME`.
2. En ***Examinar...*** buscamos la localización de la carpeta `jdk` donde lo hayamos instalado.
3. Creamos o sustituimos un **Path** con `%JAVA_HOME%\bin`.

¡Ahora sí estamos listos!

>> 1.2.3- NUESTRO PRIMER PROGRAMA EN JAVA

Basta ya de tanta charla, ¡es hora de empezar a programar! Vamos a **crear un programa simple** paso a paso y a **ejecutarlo**.

1

CREACIÓN DEL CÓDIGO FUENTE

En cualquier **editor de texto** (wordpad, notepad…) crea en el **escritorio** un archivo llamado `Saludo.java`, y escribe el código que ves a la derecha.

El **nombre del archivo** debe **coincidir exactamente** con el **nombre después de** `class`. Este es el **nombre de nuestra clase**.

Saludo.java

```
public class Saludo{
  public static void main(String[] args){
    System.out.println("¡Bienvenido a JAVA!");
  }
}
```

Sí, en vez de escribirlo, podrías copiarlo de aquí y pegarlo, pero si lo escribes a mano **comenzarás a ganar soltura** a la hora de crear código.

2

COMPILACIÓN DEL CÓDIGO FUENTE

Abre un terminal de comandos, **sitúate en el escritorio** y ejecuta la orden `javac` seguida del **nombre de tu clase** y su extensión `.java`.

Esto creará en el escritorio el **bytecode** de nuestra clase.

```
C:\Users\Diego> cd Desktop
C:\Users\Diego\Desktop> javac Saludo.java
```

Saludo.class

EJECUTAR EL BYTECODE

3 Ya solo tenemos que **ejecutar el bytecode**. Para ello, usa la orden `java` y el **nombre de la clase sin ninguna extensión**. El **resultado** de la ejecución debería mostrarse en la siguiente línea.

```
C:\Users\Diego\Desktop> java Saludo
¡Bienvenido a JAVA!
```

Recuerda que con cualquier cambio en el archivo este se debe **volver a compilar** el código fuente para **actualizar el bytecode** y que al ejecutarlo aparezcan los cambios.

1.3 - ELEMENTOS DE UN PROGRAMA JAVA

>> 1.3.1- ESTRUCTURA

Dentro de un programa JAVA podemos encontrar los siguientes elementos:

EJEMPLO

Ejemplo.java

```
1  package introduccionProgra; 1
2
3  import java.lang.System.out; 2
4
5  public class Ejemplo{ 3
6
7     public static void main(String[] args){ 4
8
9        String frase = "Hola"; 5
10
11       System.out.println(frase); 6
12       imprimirAdios();
13    }
14
15    //Método para imprimir la palabra "Adiós" 8
16    public static void imprimirAdios(){ 7
17       System.out.println("Adiós");
18    }
19 }
```

1	**PAQUETE**	Un **paquete**, **package** o **librería** es una **agrupación de clases**. Es parecido a una "caja" que contiene las clases que queramos agrupar en un solo lugar. Sólo debemos importarlas si nuestro programa necesita usar esas clases.
2	**LIBRERÍAS EXTERNAS**	Las **librerías externas** son paquetes que contienen clases con métodos útiles o necesarios para nuestra clase. Para usarlas debemos importarlas.
3	**CLASE**	Java puede crear diferentes **tipos de clases**: `public` `private` `protected` Se utilizan de acuerdo a conveniencia de la estructura de nuestro programa. El nombre de la **clase pública** debe **coincidir con el nombre del archivo**, y se **pone con la primera letra en mayúscula**.
4	**MÉTODO `main()`**	Un programa en Java se puede considerar una **colección de clases**, en la que **al menos una de ellas incluya de manera obligatoria al método `main()`**. Este

		método indica el comienzo del programa y es el lugar donde se ejecutarán las instrucciones del mismo, incluyendo llamadas a métodos.
	Las clases y los métodos, después de su nombre, deberán llevar una llave de apertura (`{`) que indica el comienzo y una llave de cierre (`}`) para indicar el fin.	
5	**ZONA DE DECLARACIÓN DE VARIABLES**	Java maneja tres tipos de variables: de **instancia**, de **clase** y **locales**. **INSTANCIA**: Son las que se usan para **guardar valores o atributos de un objeto en particular**. **CLASE**: Son las que **guardan valores o atributos de la clase**. **LOCALES**: Son las que se **declaran en una función o método** y solamente **las puede utilizar esa función o método**.
6	**CUERPO**	Compuesto por **sentencias e instrucciones de código**. **Todas las sentencias deben acabar en punto y coma (`;`), excepto la declaración de métodos y clases.**
7	**MÉTODOS**	Son definidos por el usuario dentro de las clases. En Java, los **métodos son los que utilizamos para realizar alguna tarea en específico**.

8	**COMENTARIOS**	Éstos pueden incluirse en **cualquier parte del código**. Sus líneas serán **completamente ignoradas por el compilador**, o sea que no afectarán para nada nuestro programa. Se usan para **aclarar ciertas partes del código** o **proporcionar información importante junto a este**.
		`//Los comentarios se pueden crear así, con barras dobles al inicio.`
		`/* O separarlos en varias líneas`
		`de esta manera, con barras y asteriscos. */`
		`/**`
		`* Estos comentarios forman parte de un`
		`*/`

>> 1.3.2- IDENTIFICADORES

En los lenguajes de programación, los **identificadores**, como su nombre lo indica, se utilizan **con fines de identificación**. En Java, un identificador puede ser un nombre de clase, un nombre de método o un **nombre de variable**.

Cuando usamos identificadores en JAVA (aunque esto también puede aplicarse a C y C++), debemos tener en cuenta unas **reglas básicas para poder definirlos**:

Los **únicos caracteres permitidos** son **caracteres alfanuméricos** (`A-Z`, `a-z` y `0-9`), el signo de dólar (`$`) y el guión bajo (_). No se permiten los **espacios** ni **caracteres especiales** (incluyendo las **tildes**).	✖ `primera clase` ✔ `primeraClase` ✔ `primera_clase` ✖ `salón` ✔ `salon`
Es recomendable sustituir la `ñ` por `nn` ó `ny`.	✖ `año` ✔ `anno` ✔ `anyo`
Los identificadores **no deben comenzar con dígitos** (`0-9`).	✖ `9hola` ✔ `hola9`
Los identificadores de Java distinguen entre mayúsculas y minúsculas.	`anyo_a;` ✖ `anyo_A = ...;` ✔ `anyo_a = ...;`

No hay límite en la longitud del identificador, pero es aconsejable usar solamente una **longitud óptima de 4 a 15 caracteres**.	
Las **palabras reservadas no se pueden usar como un identificador**, **ni los valores lógicos `true` o `false`.** Aunque se cambie su sintaxis, **no es recomendable usarlos** tampoco.	✖ `public` ✔ `Public`
No pueden ser iguales a otro identificador declarado en el mismo ámbito.	`anyo_a;` ✖ `anyo_a;`

Otras reglas aplicables son las **impuestas por conveniencia**:

Los **nombres de las variables y los métodos** deberían **empezar por una letra minúscula** y los de las **clases por mayúscula**. Las **constantes** deberían **estar escritas enteras en mayúsculas**.	✔ `miVariable` ✔ `MiClase` ✔ `MICONSTANTE`
Si el identificador está formado por **varias palabras**, **la primera se escribe en minúsculas (excepto para las clases)** y el **resto de palabras se hace empezar por mayúscula (camel case)** o estar **separadas por un guión** (**snake case**).	✔ `numeroDeCurso` ✔ `numero_de_curso`

Es de vital importancia elegir **nombres adecuados para los identificadores**. El motivo es **facilitar la lectura del código todo lo posible**, haciendo que con un simple vistazo **sepamos lo que hace**. Identificadores como `a`, `b`, `c`, `objeto`, `uno`, ... etc. **no son adecuados** porque **no indican nada acerca del elemento identificado**.

También es importante que si comienzas a usar una **nomenclatura**, **mantengas esta a lo largo de todo el código**. No empieces usando **camel case** y uses a veces **snake case** en un mismo programa. **Esto no queda bien visualmente**.

1.4 - TIPOS DE DATOS

En Java existen dos tipos principales de datos:

SIMPLES	COMPUESTOS
Nos permiten crear **variables que almacenan un solo valor**. Son los que más vamos a utilizar por ahora.	Permiten **almacenar muchos datos**. Los veremos en futuras unidades.

Aunque lo veremos más adelante, existe un tipo de dato compuesto llamado `String` que conviene conocer ya que permite **representar texto**.

>> 1.4.1- DATOS SIMPLES

Los datos simples nos permiten almacenar lo siguiente:

Números enteros y reales	Caracteres	Valores lógicos

TIPO	¿QUÉ ES?	MEMORIA EN BITS	RANGO DE VALORES	VALOR POR DEFECTO
`byte`	Núm. entero	8	`-128 ... 127`	0

short	Núm. entero	**16**	-32768 ... 32767	0
int	Núm. entero	**32**	-2147483648 ... 2147483647	
long	Núm. entero	**64**	-9223372036854775808 ... 9223372036854775807	0
float	Núm. real de precisión simple	**32**	$\pm 3,4*10^{-38}$ ... $\pm 3,4*10^{38}$	0.0
double	Núm. real de precisión doble	**64**	$\pm 1,7*10^{-308}$ ... $\pm 1,7*10^{308}$	0.00
char	Carácter	**16**		'\u0000'
boolean	Valor lógico	**1**	true / false	false

JAVA no realiza comprobación de rangos. Si **sobrepasamos el límite de un tipo de dato**, el resultado **da la vuelta al rango y empieza de nuevo por el límite contrario**. Esto se conoce como **overflow** (desbordamiento).

***EJEMPLO*:** Si tenemos una **variable de tipo short** cuyo valor es `32.767` y se le suma `1`, el resultado será `-32.767`.

1.5 - VARIABLES

Las **variables** son **campos en los que se almacenan valores** que pueden **cambiar su valor con el tiempo**.

>> 1.5.1- DECLARACIÓN DE VARIABLES

Para **declarar una variable** se sigue la siguiente sintaxis:

```
tipo nombreVariable [= valor];
```

`tipo`	Definimos el **tipo de la variable**, usando los que ya hemos visto.
`nombreVariable`	Damos un **nombre identificativo** a la misma. Procura seguir las reglas que hemos estudiado.
`[= valor]`	Opcionalmente podemos **inicializarla** dándole un **valor inicial**. Si no, también podemos **llamar a la variable más adelante** y **asignarle allí el valor**.

EJEMPLO

Ejemplo.java

```
1  public class Ejemplo{
2
3     public static void main(String[] args){
4
5        int numero1 = 25;
6        int numero2;
7
8        numero2 = 14;
9     }
```

```
10  }
```

Si tenemos **varias variables del mismo tipo**, podemos **declararlas todas en la misma línea tras el tipo** e incluso **inicializarlas**

EJEMPLO

.java

```
1  int numero1, numero2 = 2, numero3 = 3;
```

Si una variable **no ha sido inicializada**, Java **le asigna un valor por defecto**, el cual puedes ver en la tabla de los tipos de datos **(apartado 1.41 - *Datos simples*)**.

>> 1.5.2- CONSTANTES

Si queremos **declarar una variable que nunca vaya a cambiar su valor** podemos utilizar la palabra reservada **`final`**, que además hace que **este no pueda modificarse**.

```
final tipo NOMBRECONSTANTE = valor;
```

EJEMPLO

Ejemplo.java

```
1  public class Ejemplo{
2
3     public static void main(String[] args){
4
5        final double GRAVEDAD = 9.8;
6        double fuerza;
7        double masa;
8
9        fuerza = masa * GRAVEDAD;
10
11    }
12 }
```

Si intentamos **modificar el valor de la constante**, el compilador **lanzará un error.**

EJEMPLO

Ejemplo.java

```
1  public class Ejemplo{
2
3     public static void main(String[] args){
4
5        final double GRAVEDAD = 9.8;
6        double fuerza;
7        double masa;
8
9        GRAVEDAD = 10;
10
11       fuerza = masa * GRAVEDAD;
12
13    }
14 }
```

1.6 - OPERADORES

Los **operadores** son una parte **indispensable** de la programación ya que nos permiten **realizar cálculos matemáticos y lógicos**, entre otras cosas. Los operadores pueden ser:

ARITMÉTICOS	RELACIONALES	LÓGICOS	DE ASIGNACIÓN

>> 1.6.1- OPERADORES ARITMÉTICOS

Se utilizan en datos tipo `NUMBER`, englobando **las cifras del** `0` **al** `9`, **punto decimal y signo negativo**.

		EJEMPLO	RESULTADO
`+`	**Suma**.	`4+7`	`7`
`-`	**Resta**.	`5-3`	`2`
	Cambio de signo.	`-6`	`6`
`*`	**Multiplicación**.	`3*2`	`6`
`/`	**División entera**.	`6/2`	`3`
`%`	**Resto de división entera**.	`20%7`	`6`
`++`	**Incremento unitario** (equivale a `x = x+1`).	`5++` `++5`	`6` `6`
`--`	**Decremento unitario** (equivale a `x = x-1`).	`5--` `--5`	`4` `4`

Los **operadores incrementales** suelen utilizarse a menudo en **bucles** (estructuras repetitivas) que veremos más adelante.

>> 1.6.2- OPERADORES RELACIONALES

Los **operadores relacionales** actúan sobre **valores enteros**, **reales** y **caracteres** (`char`); y devuelven un valor del tipo `boolean` (`true` o `false`).

		EJEMPLO	RESULTADO
`>`	Devuelve `true` si `a` es mayor que `b`.	`5>3`	`true`
`<`	Devuelve `true` si `a` es menor que `b`.	`5<3`	`false`
`>=`	Devuelve `true` si `a` es mayor o igual que `b`.	`5>=3`	`false`
`<=`	Devuelve `true` si `a` es menor o igual que `b`.	`5<=5`	`true`
`==`	Devuelve `true` si `a` es igual que `b`.	`5==5`	`true`
`!=`	Devuelve `true` si `a` es distinto que `b`.	`4!=3`	`true`

>> 1.6.3- OPERADORES LÓGICOS

Estos operadores actúan sobre **operadores o expresiones lógicas**, es decir, **aquellos que se evalúan a `true` o `false`**.

`&&`	**Y** lógico. Devuelve `true` si `a` y `b` son `true`.
`||`	**O** lógico. Devuelve `true` si `a` o `b` es `true`.
`!`	**Negación** lógica. Devuelve `true` si `a` es `false`.

Aquí puedes ver las combinaciones de ambos:

a	b	a && b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

a	b	a || b
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

a	!a
TRUE	FALSE
FALSE	TRUE

>> 1.6.4- OPERADOR DE ASIGNACIÓN

`=`	A través de este símbolo **asignamos a la variable el resultado** de evaluar **la expresión de la derecha**.

Es posible **combinar** el operador de **asignación** con **otros operadores** para, de forma abreviada, **realizar un cálculo y asignarlo a una variable**.

	FORMATO	EQUIVALENCIA
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`
`&=`	`a &= b`	`a = a & b`
`|=`	`a |= b`	`a = a | b`
`^=`	`a ^= b`	`a = a ^ b`

>> 1.6.5- ORDEN DE PREFERENCIA DE OPERADORES

1	**Postfijos**	`[]`
2	**Unarios**	`++x  --x  -x  ~!`
3	**Creación o conversión de tipo**	`new (tipo) x`
4	**Multiplicación y división**	`*  /  %`
5	**Suma y resta**	`+  -`
6	**Desplazamiento de bits**	`<<  >>  >>>`

7	Relacionales	< > <= >=
8	Igualdad y negación	== !=
9	AND (Bits)	&
10	AND Lógico	&&
11	XOR (Bits)	^
12	OR (Bits)	|
13	OR Lógico	||
14	Condicional	? :
15	Asignación	+= -= *= /= %= &= |= ^= >>= <<=

¿Qué pasa si queremos hacer **operaciones más complejas** como **potencias** o **raíces cuadradas**? Echa un vistazo al **Anexo: Implementación y uso de métodos y clases recomendadas - *Clase `Math`***.

1.7 - LITERALES

A la hora de tratar con **valores** de los **tipos de datos simples** (y `String`) se utiliza lo que se denomina **literales**. Los literales son **elementos que sirven para representar un valor en el código fuente del programa**.

LÓGICOS	ENTEROS	DECIMALES	CARÁCTER	CADENAS DE CARACTERES
`boolean`	`byte short int long`	`double float`	`char`	`String`

>> 1.7.1- LITERALES LÓGICOS

Los literales **lógicos** son:

`true`	`false`

Y ya está :).

>> 1.7.2- LITERALES ENTEROS

Los literales **numéricos enteros** se consideran como tipo `int`.

CONVERSIONES (CASTING) IMPLÍCITAS CON ENTEROS

Dado que todo **literal entero** es un tipo `int`, la siguiente operación podría provocar un **error de compilación**, ya que intentaría asignar un número de tipo `int` a una variable de tipo `byte`:

```
byte b = 10;
```

Sin embargo, como veremos más adelante, en expresiones de este tipo Java realiza **una conversión implícita del dato al tipo destino**, siempre y cuando **el dato "quepa" en la variable**.

Puedes saber más sobre casting en el **Anexo: Implementación y uso de métodos y clases recomendadas - *Conversión de tipos: Casting implícito y explícito***.

>> 1.7.3- LITERALES DECIMALES

Los literales de tipo **decimal** sirven para **indicar valores `float` o `double`**, pero por defecto es un tipo **`double`**.

CONVERSIÓN DE DOUBLE A FLOAT

Una asignación del siguiente tipo provoca un **error de compilación** al intentar asignar un dato **`double`** a una variable **`float`** que tiene un tamaño menor:

```
float p = 3.14;
```

Para evitar el error, se debe utilizar la letra **`f` a continuación del número**, lo que provoca una conversión del número **`double`** a **`float`**.

```
float p = 3.14f;
```

>> 1.7.4- LITERALES DE CARÁCTER

Los literales de tipo **carácter** se representan siempre entre **comillas simples (`'`)**. Entre estas puede aparecer:

VALOR UNICODE HEXADECIMAL	SÍMBOLOS UNICODE	ENTEROS PARA VALORES DE CARACTERES UNICODE
`'\U03AF'`	`'a'` `'B'` `'é'` `'{'`	`'231'`

SECUENCIAS DE ESCAPE
Las secuencias de escape generan un **comportamiento específico** en los caracteres según el **comando** que se escriba.

Nueva línea	Tabulador	Retroceso de carro	Comienzo de página	Borrado a la izquierda	Barra inversa	Prima simple	Prima doble
`'\n'`	`'\t'`	`'\r'`	`'\f'`	`'\b'`	`'\\'`	`'\''`	`'\"'`
					Estos tipos imprimen el **carácter al lado de la barra diagonal**.		
					`\`	`'`	`"`

>> 1.7.5- LITERALES DE CADENAS DE CARACTERES (`STRING`)

Los **`Strings`** o **cadenas de caracteres no forman parte de los tipos de datos elementales en Java**, sino que son **instanciados a partir de la clase `java.lang.String`**, pero aceptan su inicialización a partir de literales de este tipo.

Un **literal** de tipo **`String`** va encerrado entre **comillas dobles (`"`)** y debe estar incluido **completamente en una sola línea del programa fuente** (no puede dividirse en varias líneas).

Entre las comillas dobles puede incluirse **cualquier carácter del código Unicode (o su código precedido del carácter \ )** además de las secuencias de escape vistas en el **apartado 1.7.4 - *Literales Carácter***. Así, por ejemplo, para incluir un **cambio de línea dentro de un literal de tipo `String`**, lo hacemos mediante la secuencia de escape **`\n`** :

EJEMPLO

Ejemplo.java

```java
public class Ejemplo{
   public static void main(String[] args){
      System.out.println("Primera línea\nSegunda línea");
      System.out.println("Hola");
   }
}
```

Console

```
Primera línea
Segunda línea
Hola
```

También podemos **concatenar `Strings`** a través del operador **+** en **diferentes líneas del código**.

EJEMPLO

Ejemplo.java

```java
public class Ejemplo{
   public static void main(String[] args){
      System.out.println("Como este texto es muy largo para la pantalla lo "
      + "que hago es dividirlo en dos.");
   }
}
```

Console

```
Como este texto es muy largo para la pantalla lo que hago es dividirlo en dos.
```

Los **`Strings` no pueden compararse con operadores normales**, por lo que para ello debemos usar el método **`.equals()`**.

EJEMPLO

Ejemplo.java

```java
public class Ejemplo{
   public static void main(String[] args){
      String textoOriginal = "abc";
      String textoAComparar = "ABC";

      System.out.println(textoOriginal.equals(textoAComparar));
   }
}
```

Console

```
false
```

Podemos **ignorar las mayúsculas y minúsculas** con **`.equalsIgnoreCase()`**.

EJEMPLO

Ejemplo.java

```java
public class Ejemplo{
   public static void main(String[] args){
```

```java
        String textoOriginal = "abc";
        String textoAComparar = "ABC";

        System.out.println(textoOriginal.equalsIgnoreCase(textoAComparar));
    }
}
```

Console

```
true
```

1.8 - SALIDA Y ENTRADA ESTÁNDAR

¿Cómo hacer que la **información salga y entre por pantalla**? Vamos a explorarlo un poco:

>> 1.8.1- SALIDA ESTÁNDAR

· 1.8.1.1- SALIDA DE LÍNEAS DE STRING

Ya hemos visto en algún ejemplo la línea `System.out` para sacar cosas por consola. Tenemos tres métodos útiles muy parecidos para ello:

`.print(datos...)`	**Imprime por pantalla** los datos que pasemos.
`.println(datos...)`	**Imprime por pantalla** los datos que pasemos **en una nueva línea**.
`.err(datos...)`	**Imprime por pantalla** y **en color rojo** los datos que pasemos. Se usa para proporcionar información sobre un **error**.

EJEMPLO

Ejemplo.java

```java
public class Ejemplo{
   public static void main(String[] args){
      System.out.print("¡Hola");
      System.out.println(" a todos!");
      System.out.println("¿Qué tal?");
      System.out.err("Os presento un error");
   }
}
```

Console

```
¡Hola a todos!
¿Qué tal?
Os presento un error
```

· 1.8.1.2- SALIDA DE LÍNEAS CON DATOS DE VARIABLES

También pueden imprimirse **variables de cualquier tipo**, así como **combinaciones de texto y variables concatenadas con el operador +**.

EJEMPLO

Ejemplo.java

```java
public class Ejemplo{
    public static void main(String[] args){

        String nombre = "Juan";
        int edad = 22;

        System.out.print(nombre);
        System.out.println(edad);
        System.out.println(nombre + " tiene " + edad + " años.");
    }
}
```

Console

```
Juan
22
Juan tiene 22 años.
```

>> 1.8.2- ENTRADA ESTÁNDAR

Si queremos que el usuario **introduzca su propia información**, haremos uso de un `Scanner`. Es un proceso un poco complejo pero no muy extenso.

· 1.8.2.1- OBJETO SCANNER

Para poder usar pedir datos por teclado, debemos declarar un objeto de tipo Scanner que los recoja. Antes de poder declararlo, debemos **importar la clase** `Scanner`.

```java
import java.util.Scanner;

Scanner nombreScanner = new Scanner(System.in);
```

`Scanner`	El **tipo** de nuestro lector será `Scanner`.
`nombreScanner`	Damos un **nombre al** `Scanner`. Se suele usar `reader` o `keyboard`, o cualquier **nombre que lo identifique como tal**.
`= new Scanner(System.in);`	La **inicialización del objeto**. Veremos esto más a detalle con **otros objetos** en las próximas unidades.

Una vez declarado, sólo debemos **asignar a una variable** que recoja el dato nuestro objeto `Scanner` con un método que defina el tipo de dato. Sin la asignación, el dato se **perderá**.

Normalmente, JAVA te pedirá **cerrar** el `Scanner` cuando ya no se utilice, pero como normalmente nosotros lo usaremos siempre, no nos hace falta cerrarlo. Aún así, no es mala práctica abrir el `Scanner`, escribir nuestro código dentro del mismo y colocar al final la **sentencia de cierre**:

```java
nombreScanner.close();
```

Quedando el código de la **siguiente manera**:

```java
Scanner nombreScanner = new Scanner(System.in);{
        instrucciones...
nombreScanner.close();}
```

Ten en cuenta que **a partir de la sentencia de cierre** el `Scanner` **ya no funcionará**.

· 1.8.2.2- RECOGIDA DE DATOS A TRAVÉS DE UN SCANNER

¿Cómo sabemos qué tipo de dato debemos recoger? Dependiendo de la variable donde vayamos a recoger esta información, deberemos usar un método del **`Scanner`** u otro. Estos son los principales:

`.nextByte()`	Obtiene un número entero de tipo **`byte`**.
`.nextShort()`	Obtiene un número entero de tipo **`short`**.
`.nextInt()`	Obtiene un número entero de tipo **`int`**.
`.nextLong()`	Obtiene un número entero de tipo **`long`**.
`.nextFloat()`	Obtiene un número real de tipo **`float`**.
`.nextDouble()`	Obtiene un número real de tipo **`double`**.
`.nextLine()`	Registrará un **`String`** hasta que se introduzca la tecla **Enter**. También es útil para recoger **Enters perdidos** que se generan al **recoger datos numéricos**.
`.next()`	Obtiene el siguiente **token** (un **`String`** hasta **un carácter espacio**).
`.nextBoolean()`	Obtiene el siguiente **token** (un **`String`** hasta **un carácter espacio**) y lo transforma en un **valor booleano**.

EJEMPLO

Ejemplo.java

```java
import java.util.Scanner;

public class Ejemplo{
   public static void main(String[] args){

      Scanner keyboard = new Scanner(System.in);{

         String palabra;
         int edad;

         System.out.println("Introduce una palabra:");
         palabra = keyboard.nextLine();

         System.out.println("Introduce tu edad:");
         palabra = keyboard.nextInt();

         System.out.println("Tu palabra: " + palabra);
         System.out.println("Tu edad: " + edad);

      keyboard.close();}
   }
}
```

Console

```
Introduce una palabra:
Hola
Introduce tu edad:
25
Tu palabra: Hola
Tu edad: 25
```

En cuanto el flujo de código llega a un uso del **`Scanner`**, el programa **se para** hasta que el usuario introduce la información a través del teclado.

U1 - **BATERÍA DE EJERCICIOS**

>> PARTE 1

1	Diseña un algoritmo en pseudocódigo que calcule el área de un cuadrado y lo muestre.
2	Diseña un algoritmo en pseudocódigo que lea dos números, calcule y muestre el valor de su suma, resta, producto y división.
3	Diseña un algoritmo en pseudocódigo que dado el precio de un producto aplique el porcentaje de descuento que debe aplicarse y muestre el precio final.
4	Diseña un algoritmo en pseudocódigo para calcular el número de horas que hay en 10 años.
5	Diseña un algoritmo en pseudocódigo que lee dos números y muestra el mayor.
6	Diseña un algoritmo en pseudocódigo que te pida tu edad e indique si eres mayor de edad o no.
7	Diseña un algoritmo en pseudocódigo que lee dos números y muestra el mayor o si son iguales.
8	Dado el siguiente algoritmo descrito en pseudocódigo, explica brevemente qué hace y cuál sería el resultado mostrado si el valor R leído fuera 2. *Inicio* *Leer R* *A = 3,14 * R * R* *Mostrar A* *Fin*
9	Escribe un programa que muestre tu nombre por pantalla.
10	Modifica el programa anterior para que además se muestre tu dirección y tu número de teléfono. Asegúrate de que los datos se muestran en líneas separadas.
11	Escribe un programa que muestre por pantalla 10 palabras en inglés junto a su correspondiente traducción al castellano. Las palabras deben estar distribuidas en dos columnas y alineadas a la izquierda. Pista: Se puede insertar un tabulador mediante `\t`.
12	Escribe un programa que pinte por pantalla una pirámide rellena a base de asteriscos. La base de la pirámide debe estar formada por 9 asteriscos.
13	Igual que el programa anterior, pero esta vez la pirámide estará hueca (se debe ver únicamente el contorno hecho con asteriscos).

>> PARTE 2

1

Dado el siguiente listado de **identificadores**, indicar si son correctos o no (justifica la respuesta):

`variable`	`variable*`
`mi edad`	`edadAddulto`
`1_parametro`	`parametro-1`
`parametro_2`	`$valor`
`valorA`	`a`
`int`	`recuento_total`
`recuentoTotal`	`suma`
`a+b`	`123_valor`
`class`	`porcentaje%`

2

La siguiente tabla muestra un algoritmo paso a paso (lista de instrucciones). Utiliza tres variables **`A`**, **`B`** y **`C`** que inicialmente valen **`4`**, **`2`** y **`3`** respectivamente. Calcula el valor de las variables tras ejecutar cada instrucción. Las tres primeras están hechas a modo de ejemplo.

Instrucción	A	B	C
Valores iniciales	4	2	3
`A = B`	2	2	3
`C = A`	2	2	2
`B = (A + B + C) / 2`	2	3	2
`A = A + C`			
`C = B - A`			
`C = C - A`			
`A = A * B`			
`A = A + 3`			
`A = A % B`			
`C = C + A`			

3

Evalúa las siguientes expresiones:

```
((3 + 2) ^ 2 - 15) / 2 * 5
```

```
5 - 2 > 4 AND NOT 0.5 == 1 / 2
```

```
Dado x = 1, y = 4, z = 10, pi = 3.14, e = 2.71
2 * x + 0.5 + y - 1 / 5 * z
```

```
Dado x = 1, y = 4, z = 10, pi = 3.14, e = 2.71
pi * x ^ 2 > y OR 2 * pi * x <= z
```

El símbolo **`^`** se utiliza para indicar potencia **`2^3`** es equivalente a **`23`** en JAVA si queremos usar potencias hay que usar la función **`Math.pow(2,3)`**. Por tanto si se quiere probar este ejercicio en JAVA hay que hacer ese cambio.

4

Indica qué palabras de las siguientes no son tipos básicos del lenguaje Java:

`int` **`double`** **`Long`** **`boolean`** **`byte`** **`char`** **`string`**

5

Crear un programa en el que se declaren dos variables de tipo entero (**`int`**), asignándole valor **`20`** y **`4`** respectivamente, y muestre el resultado de la suma, resta, multiplicación, división y módulo.

6	Crear un programa, que indique si un número es mayor que otro.
7	Crear un programa que muestre el área de un cuadrado de lado **5**.
8	Crear un programa que calcule y muestre el número de horas que hay en 10 años.
9	Crear un programa que realice las instrucciones del ejercicio 2, y muestre el valor final de las tres variables por pantalla.
10	Crear un programa que muestre el área de un cuadrado cuyo lado se introduce por teclado.
11	Crear un programa en el que se declara la variable **`int valor = 1`**, y muestre por pantalla los números del 1 al 10 utilizando el incremento unitario.
12	Crear un programa que muestre el seno, coseno y tangente de **100**. (Utilizar la clase **Math**)
13	Crear un programa dados dos números muestre cual es el mayor. (Se puede utilizar la clase **Math**)

14	El siguiente código tiene un error de compilación. Indica cuál es:
	`int k = 2500;`
	`float r = 3.7;`
	`byte p = (byte) k;`

15	Teniendo en cuenta que **a**, **b** y **c** son variables de tipo **int** que han sido inicializadas con algún valor, indica cuál de las siguientes instrucciones es incorrecta: 1. `a = b*c++;` 2. `a=c^b;` 3. `a = b && c;`
16	Las variables de tipo booleano pueden guardar los resultados de operaciones lógicas. Crea un programa en el que se establezcan tres valores que representen dinero en euros: un **salario**, un **pago de impuestos** y unas **tributaciones a la Seguridad Social**. El programa debe mostrar por pantalla (con valores **true** o **false**): 1. Si se paga más a la Seguridad Social que impuestos. 2. Si el pago de impuestos y seguridad social superan la mitad del salario. 3. Si las dos afirmaciones anteriores son ciertas. 4. Si el salario o el pago de impuestos o el pago de Seguridad Social son menores de 500 euros. 5. Si el pago de impuestos o la Seguridad Social son menores de 200 euros cada uno y el salario supera los 1500.
17	Crear un conversor de tiempo. En una variable se guardará un tiempo en segundos y se debe mostrar por pantalla convertido en horas, minutos y segundos. Presta atención a crear un formato de salida agradable para el usuario final.
18	Crear un programa en el que se declaren variables de tipo **float** y **double**, se les asignen valores y se muestre por pantalla el valor.
19	Crear un programa en el que se declaren cuatro variables de tipo **char**: **c**, **a**, **s**, **a** y con ellas se muestre la palabra '**casa**' por pantalla.
20	Crear un programa que solicite nombre y edad y los muestre por pantalla.
21	Crear un programa que solicite nombre y edad en la misma línea y se introduzcan separados por un espacio y los muestre por pantalla.
22	Crear un programa que solicite tres números (pueden tener decimales) y muestre el resultado de hacer la media de los tres.
23	Crear un programa que muestre por pantalla el mensaje **ERROR!!!**. Este debe salir en rojo como los mensajes de error.