



Programación

# UD02. Introducción a Java



Materiales elaborados por:  
Anna Sanchis Perales  
Ampliados y modificados  
Joan Carrillo


## Licencia





**Reconocimiento – NoComercial - CompartirIgual (BY-NC-SA):** No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

## Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante

 Atención

 Interesante

## ÍNDICE DE CONTENIDO

<b>1. Introducción</b>	<b>4</b>
1.1 ¿Qué es Java?	4
¿Por qué usar Java?	4
¿Cuándo no usar Java?	5
¿Qué más aporta Java?	5
¿Qué hace falta para usar un programa creado en Java?	5
¿Qué hace falta para crear un programa en Java?	6
<b>2. Instalación del JDK</b>	<b>7</b>
<b>3. Instalación de visual Studio Code</b>	<b>8</b>
<b>4. Escribiendo “Hola Mundo”</b>	<b>9</b>
4.1 Entendiendo la primera aplicación	13
<b>5. Variables</b>	<b>17</b>
5.1 Qué son y cómo se declaran las variables	17
5.2 Operaciones matemáticas básicas.	21
5.3 Incremento y asignaciones abreviadas.	22
5.4 Operadores relacionales	23
5.5 Operadores lógicos	24
<b>6. Comprobación de condiciones</b>	<b>25</b>
6.1 if	25
6.2 else	26
6.3 switch	27
6.4 El operador condicional	29
<b>7. Partes del programa que se repiten</b>	<b>30</b>
7.1 while	30
7.2 do- while	30
7.3 for	31
7.4 break y continue	33
<b>8. Arrays y cadenas de texto</b>	<b>34</b>
6.1 Los arrays	34
6.2 Las cadenas de texto	37
<b>9. Datos introducidos por el usuario</b>	<b>41</b>
7.1 Cómo leer datos desde consola con la clase Scanner	41
7.2 Limpiar el buffer de entrada de la clase Scanner	42
<b>10. Parse</b>	<b>45</b>

## 1. INTRODUCCIÓN

### 1.1 ¿Qué es Java?

**Java es un lenguaje de programación** de ordenadores, diseñado como una mejora de C++, y desarrollado por Sun Microsystems (compañía actualmente absorbida por Oracle).

Hay varias hipótesis sobre su origen, aunque la más difundida dice que **se creó para ser utilizado en la programación de pequeños dispositivos**, como aparatos electrodomésticos (desde microondas hasta televisores interactivos). Se pretendía crear un lenguaje con algunas de las características básicas de C++, pero que necesitará menos recursos y que fuera menos propenso a errores de programación.

De ahí evolucionó (hay quien dice que porque el proyecto inicial no acabó de funcionar) hasta convertirse en **un lenguaje muy aplicable a Internet y programación de sistemas distribuidos en general**. Pero su campo de aplicación no es exclusivamente Internet: una de las grandes ventajas de Java es que se procura que sea **totalmente independiente del hardware: existe una "máquina virtual Java"** para varios tipos de ordenadores. Un programa en Java podrá funcionar en cualquier ordenador para el que exista dicha "máquina virtual Java" (hoy en día es el caso de los ordenadores equipados con los sistemas operativos Windows, Mac OS X, Linux, y algún otro; incluso muchos teléfonos móviles actuales son capaces de usar programas creados en Java). Y aún hay más: el sistema operativo **Android** para teléfonos móviles usa Java como lenguaje estándar para crear aplicaciones. Como inconveniente, la existencia de ese paso intermedio hace que los programas Java no sean tan rápidos como puede ser un programa realizado en C, C++ o Pascal y optimizado para una cierta máquina en concreto.

### ¿Por qué usar Java?

Puede interesarnos si queremos crear programas que se vayan a manejar a través de un interfaz web (sea en Internet o en una Intranet de una organización), programas distribuidos en general, o programas que tengan que funcionar en distintos sistemas sin ningún cambio (programas "portables"), o programas para un Smartphone Android, entre otros casos.

### ¿Cuándo no usar Java?

Como debe existir un paso intermedio (la "máquina virtual") para usar un programa en Java, no podremos usar Java si queremos desarrollar programas para un sistema concreto, para el que no exista esa máquina virtual. Y si necesitamos que la velocidad sea la máxima posible, quizá no sea admisible lo (poco) que ralentiza ese paso intermedio.

### ¿Qué más aporta Java?

Tiene varias características que pueden sonar interesantes a quien ya es programador, y que ya irá conociendo poco a poco quien no lo sea:

- La sintaxis del lenguaje es muy parecida a la de C++ (y a la de C).
- Al igual que C++, es un **lenguaje orientado a objetos**, con las ventajas que eso puede suponer a la hora de diseñar y mantener programas de gran tamaño.
- Java permite crear **programas multitarea**.
- Permite **excepciones**, como alternativa más sencilla **para manejar errores**, como ficheros inexistentes o situaciones inesperadas.
- **Es más difícil cometer errores** de programación que en C y C++ (no existen los punteros).
- Se pueden crear entornos "basados en ventanas", gráficos, acceder a bases de datos, etc.

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

Vamos a centrarnos en el caso de un "ordenador convencional", ya sea de escritorio o portátil.

Las aplicaciones que deban funcionar "por sí solas" necesitarán que en el ordenador de destino exista algún "intérprete" de Java, eso que hemos llamado la "**máquina virtual**". Esto es cada vez más frecuente (especialmente en sistemas como Linux), pero si no lo tuviéramos (como puede ocurrir en Windows), basta con instalar el "**Java Runtime Enviroment**" (**JRE**), que se puede descargar libremente desde Java.com

Otra forma (actualmente menos frecuente) en que podemos encontrar programas creados en lenguaje Java, es dentro de páginas Web. Estas aplicaciones Java incluidas en una página Web reciben

el nombre de "**Applets**", y para utilizarlos también deberíamos tener instalada la máquina virtual Java (podría no ser necesario si nuestro Navegador Web reconoce automáticamente el lenguaje Java, algo que no es habitual hoy en día).

### ¿Qué hace falta para crear un programa en Java?

Existen diversas herramientas que nos permitirán crear programas en Java. La más habitual es la propia que suministra Sun (ahora Oracle), y que se conoce como **JDK (Java Development Kit)**. Es de libre distribución y se puede conseguir en la propia página Web de Oracle.

El inconveniente del JDK es que puede **no incluir un editor** para crear nuestros programas, sólo las **herramientas para generar el programa ejecutable y para probarlo**. Por eso, puede resultar incómodo de manejar para quien esté acostumbrado a otros entornos integrados, como los que de Visual C#, Visual Basic o Delphi, que incorporan potentes editores.

Pero no es un gran problema, porque es fácil encontrar editores que hagan más fácil nuestro trabajo, o incluso sistemas de desarrollo completos, **como Eclipse, NetBeans o Visual Studio Code**.










## 2. INSTALACIÓN DEL JDK

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

<https://www.oracle.com/java/technologies/javase-jdk15-downloads.html>

Allí encontraremos enlaces para descargar (download) la última versión disponible.

En primer lugar, deberemos escoger nuestro sistema operativo y (leer y) aceptar las condiciones de la licencia:

Java SE Development Kit 15		
This software is licensed under the <a href="#">Oracle Technology Network License Agreement for Oracle Java SE</a>		
Product / File Description	File Size	Download
Linux ARM64 RPM Package	141.79 MB	 <a href="#">jdk-15_linux-aarch64_bin.rpm</a>
Linux ARM64 Compressed Archive	156.98 MB	 <a href="#">jdk-15_linux-aarch64_bin.tar.gz</a>
Linux Debian Package	154.77 MB	 <a href="#">jdk-15_linux-x64_bin.deb</a>
Linux RPM Package	161.99 MB	 <a href="#">jdk-15_linux-x64_bin.rpm</a>
Linux Compressed Archive	179.31 MB	 <a href="#">jdk-15_linux-x64_bin.tar.gz</a>
macOS Installer	175.46 MB	 <a href="#">jdk-15_osx-x64_bin.dmg</a>
macOS Compressed Archive	176.07 MB	 <a href="#">jdk-15_osx-x64_bin.tar.gz</a>
Windows x64 Installer	159.68 MB	 <a href="#">jdk-15_windows-x64_bin.exe</a>
Windows x64 Compressed Archive	179.24 MB	 <a href="#">jdk-15_windows-x64_bin.zip</a>

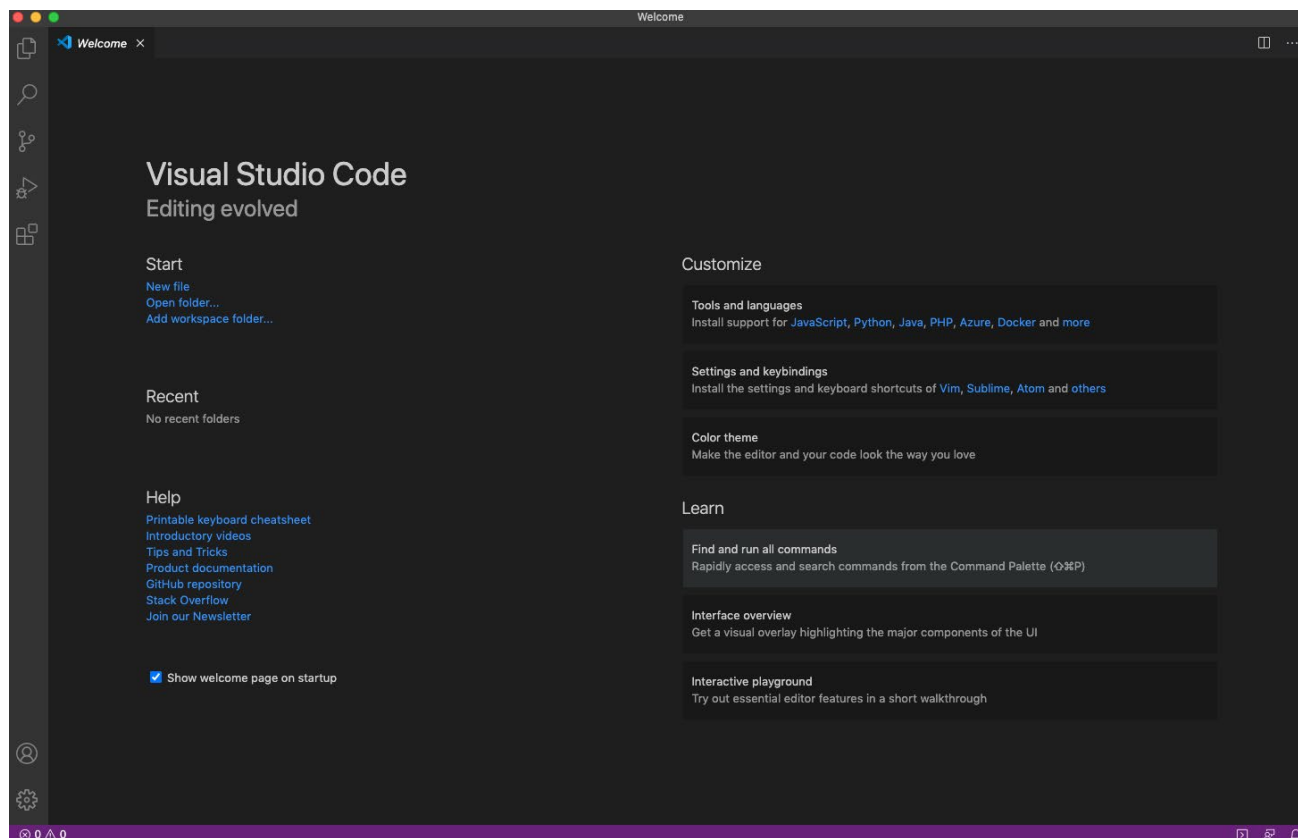
Cuando hayamos descargado, hacemos doble clic en el fichero, para comenzar la instalación propiamente dicha.

Tras ello ya disponemos del kit para hacer desarrollos en Java, a falta de un editor. Así que, ahora seguiremos con la instalación de Visual Studio Code.

### 3. INSTALACIÓN DE VISUAL STUDIO CODE

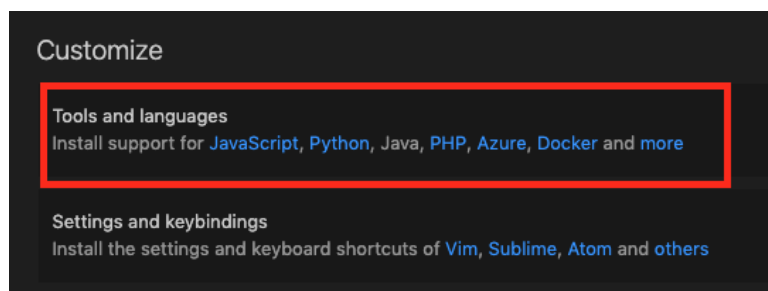
Accederemos a su web para iniciar la descarga: <https://code.visualstudio.com/>

Tras ello procederemos con su instalación. Una vez instalado procederemos a personalizarlo para que nos de soporte en el lenguaje de programación Java.



Para familiarizarnos con este IDE os recomiendo los vídeos introductorios:  
<https://code.visualstudio.com/docs/getstarted/introvideos>

Así que en el apartado “Customize” instalamos el soporte para Java.





## 4. ESCRIBIENDO “HOLA MUNDO”

Comenzaremos por crear **un pequeño programa en modo texto**. Este primer programa se limitará a escribir un texto en la pantalla.

Quien ya conozca otros lenguajes de programación, verá que conseguirlo 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.

Nuestro primer programa será:

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

A quien no conozca ningún lenguaje de programación, todo le sonará extraño, pero dentro de muy poco (apenas veamos que nuestro programa funciona) volveremos atrás para ver paso a paso qué hace cada una de las líneas que habremos tecleado.

La única línea que nos interesa por ahora es la que dice: **`System.out.println( "Hola Mundo!" );`** Esa es la orden que **se encarga de escribir Hola Mundo! en pantalla**, avanzando de línea (por eso el **"println": "print" quiere decir "escribir", y "ln" es la abreviatura de "line", línea, para indicar que se debe avanzar a la siguiente línea** después de escribir ese texto). Se trata de una **orden de salida** (out) de nuestro **sistema** (System). Esta orden es más compleja que el `PRINT` de otros lenguajes más antiguos, como BASIC, pero todo tiene su motivo, que veremos más adelante. De igual modo, a estas alturas del curso no entraremos todavía en detalles sobre qué hacen las demás líneas, que supondremos que deben existir.

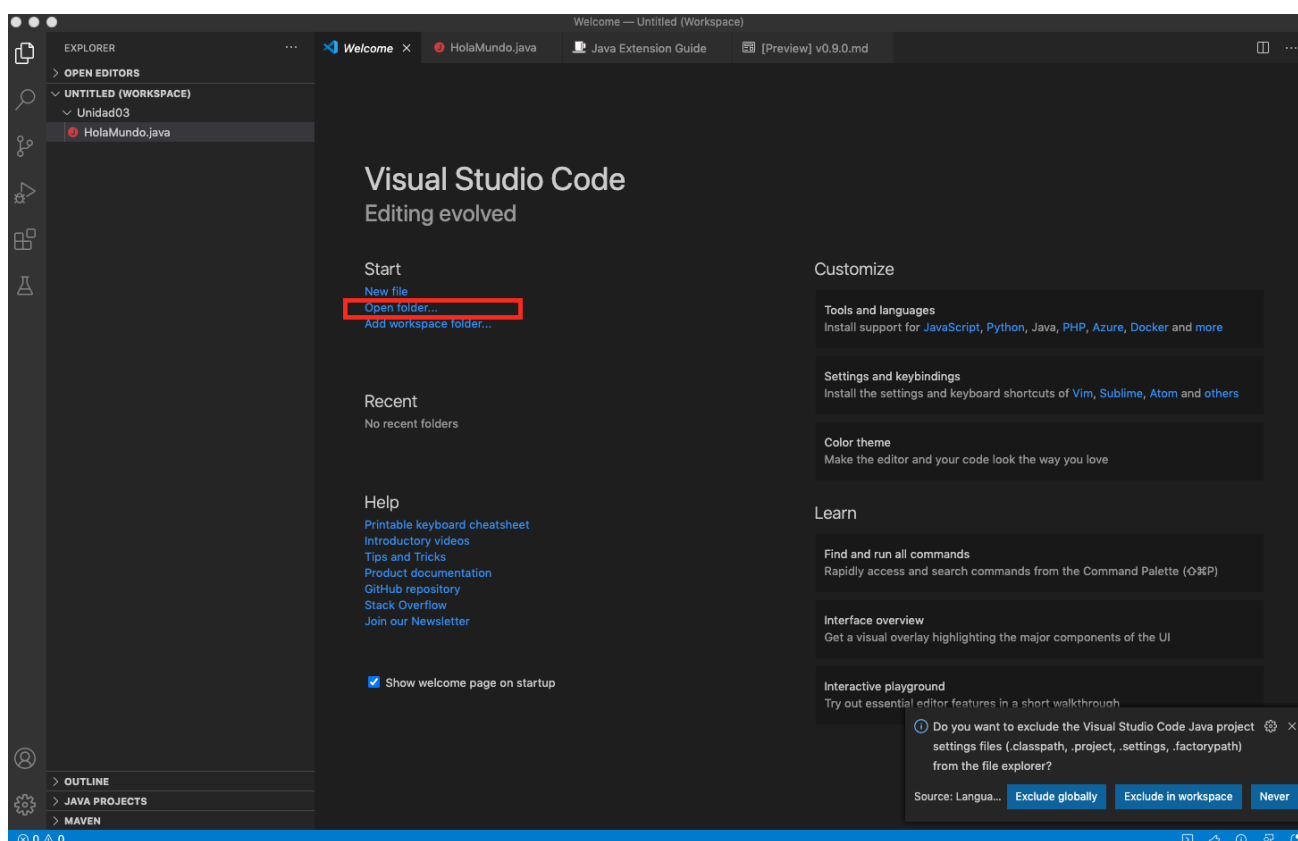
💡 El uso de mayúsculas y minúsculas es irrelevante para Windows y para algunos lenguajes,

como BASIC, pero no lo es para Java (ni para otros muchos lenguajes y sistemas). Por eso, deberemos **respetar las mayúsculas y minúsculas** tal y como aparezcan en los ejemplos, o éstos no funcionarán.

Vamos a ver qué pasos dar en **Visual Studio Code** para crear un programa como este.

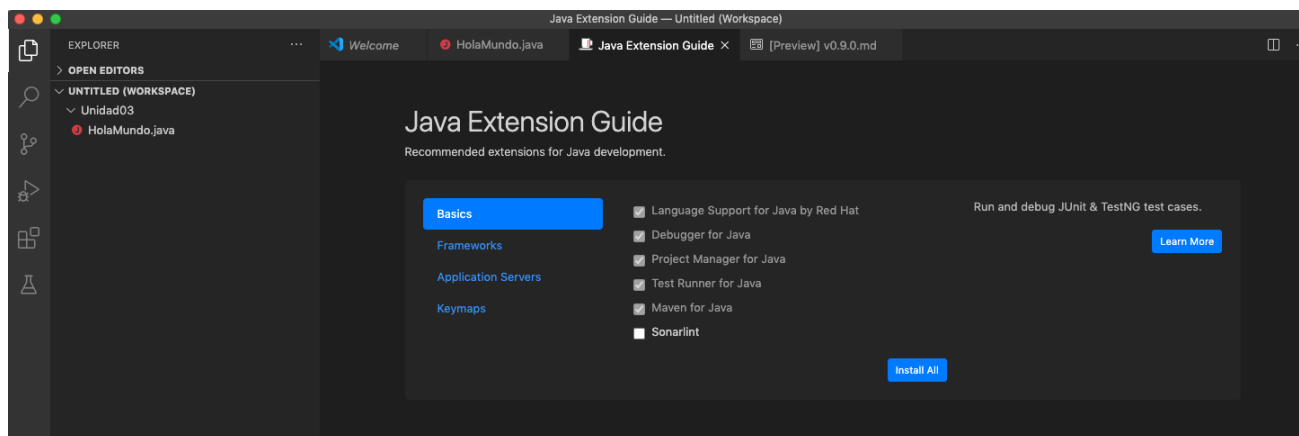
En primer lugar, deberemos crear una carpeta donde guardaremos el código Java. Una buena estructura sería crear una carpeta **CodigoJava** y dentro de ella una carpeta por cada tema, en este caso crearemos la **Unidad03**.

Tras ello abriremos la carpeta con la opción **Open folder...**



Ahora es momento de crear un nuevo fichero con la opción **File | New File** y procederemos a guardarlo con la opción **File | Save As...** le diremos **HolaMundo.java**

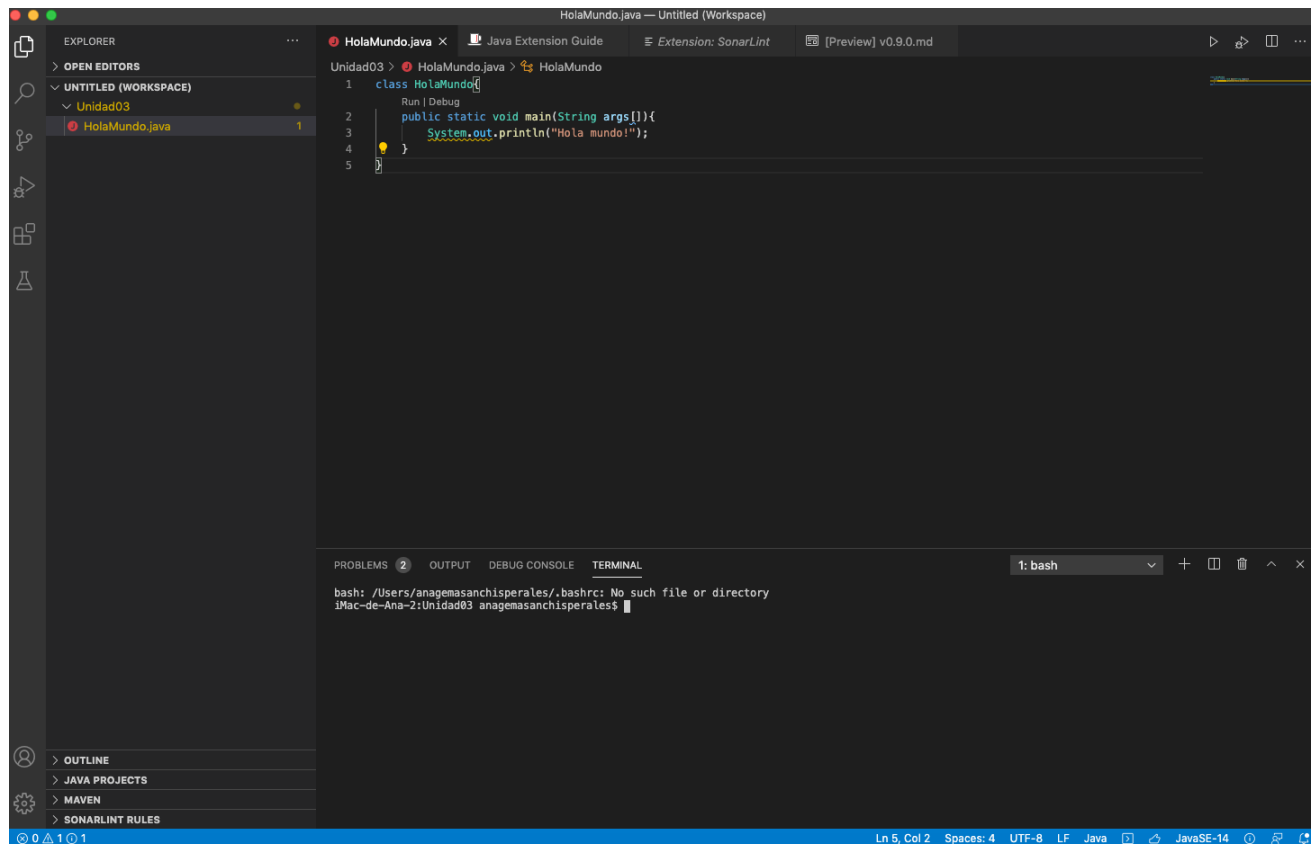
Al indicar la extensión .java se nos muestra la opción de instalar la extensión para Java, así que procedemos a instalarla. O bien, podemos acceder al apartado de extensiones y buscar el “Java Extension Pack”.



En el fichero que hemos creado copiaremos el código Java de Hola Mundo! que quedará así:

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

Es el momento de compilar el programa, lo haremos en la terminal que tiene Code, opción **Terminal** | **New Terminal**



Procedemos a compilar con la siguiente orden:

```
javac HolaMundo.java
```

Y a ejecutar:

```
java HolaMundo
```

Lo cual nos mostrará por consola el mensaje Hola Mundo!

## 4.1 Entendiendo la primera aplicación

Ahora llega el momento de entender qué hace cada línea:

El comienzo del programa: un comentario. Las tres primeras líneas son:

```
//  
// Aplicación HolaMundo de ejemplo  
//
```

Estas líneas, que comienzan con una doble barra inclinada (//) son comentarios. Nos sirven a nosotros de **aclaração**, **pero nuestro ordenador las ignora**, como si no hubiésemos escrito nada. Si queremos que un comentario ocupe varias líneas, o sólo un trozo de una línea, en vez de llegar hasta el final de la línea, podemos preceder cada línea con una doble barra, como en el ejemplo anterior, o bien indicar dónde queremos empezar con `"/"` (una barra seguida de un asterisco) y dónde queremos terminar con `"*/"` (un asterisco seguido de una barra), así:

```
/* Esta es La Aplicación HolaMundo de ejemplo */
```

**Continuamos definiendo una clase.**

Después aparece un bloque de varias órdenes; vamos a eliminar las líneas del centro y conservar solamente las que nos interesarán en primer lugar:

```
class HolaMundo {  
    [... aquí faltan más cosas ...] }
```

Esto es la **definición de una "clase" llamada "HolaMundo"**. Por ahora, podemos pensar que, a nuestro nivel de principiantes, **"clase"** es un sinónimo de **"programa"**: nuestro programa se llama HolaMundo. Más adelante veremos que los programas complejos realmente se suelen descomponer en varios bloques, planteándose como una serie de objetos que cooperan los unos con los otros. Cuando llegue ese momento, hablaremos más sobre "class" y su auténtico significado.

**¿Qué hace nuestra clase?**

Como ya hemos comentado, habrá poco más que pedir a la pantalla que muestre un cierto mensaje. Los detalles concretos de lo que debe hacer nuestra clase los indicamos entre llaves ( `{ y }` ), así :

```
class HolaMundo {  
    [... aquí faltan más cosas ...] }
```

Estas **llaves** serán frecuentes en Java, porque las usaremos para **delimitar cualquier conjunto de órdenes dentro de un programa** (y normalmente habrá bastantes de estos "conjuntos de órdenes"...)

Java es un lenguaje de formato libre, de modo que podemos dejar más o menos espacios en blanco entre los distintas palabras y símbolos, así que la primera línea de la definición de nuestra clase se podría haber escrito más espaciada

```
class HolaMundo    {  
    [... aquí faltan más cosas ...]  
}
```

o bien así (formato que prefieren algunos autores, para que las llaves de principio queden justo encima de las de final)

```
class HolaMundo  
{  
    [... aquí faltan más cosas ...]  
}
```

o incluso cualquier otra "menos legible":

```
class HolaMundo  
    {  
    [... aquí faltan más cosas ...]    }
```

Volvamos a nuestro programa...

Nuestra clase "HolaMundo" sólo contiene una cosa: un bloque llamado "**main**", que representa **el cuerpo del programa**.

```
public static void main( String args[] ) {  
    [... aquí faltan más cosas ...]  
}
```

Pero vemos que hay muchas "cosas" rodeando a "main" (public, static, void). Por ahora, simplemente

asumiremos que estas "cosas" deberán estar siempre, y más adelante volveremos a ellas según las vayamos necesitando. Lo mismo ocurrirá con eso de "String args[]": ya veremos para qué sirve y en qué circunstancias nos puede resultar útil.

Al igual que decíamos de la clase, todo el conjunto de "cosas" que va a hacer esta función "main" se deberá englobar entre llaves:

```
public static void main( String args[] ) {  
    [... aquí faltan más cosas ...] }
```

En concreto, nuestra clase de objetos "HolaMundo" interacciona únicamente con la salida en pantalla de nuestro sistema (System.out), para enviarle el mensaje de que escriba un cierto texto en pantalla (println):

```
System.out.println( "Hola Mundo!" );
```

Como se ve en el ejemplo, el texto que queremos escribir en pantalla se debe indicar entre comillas.

También es importante el punto y coma que aparece al final de esa línea: cada orden en Java deberá terminar con punto y coma (nuestro programa ocupa varias líneas pero sólo tiene una orden, que es "println").

Ahora ya podemos volver a leer todo nuestro programa, con la esperanza de entenderlo un poco más...

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

Recordemos las "cosas" importantes que no se deben olvidar de este programa:

- Nuestra **clase** (por ahora es lo mismo que decir "nuestro programa") se llama **HolaMundo**.

- Sólo tiene un **bloque**, el único que (casi) siempre aparecerá, que se llama "**main**", y que **corresponde al cuerpo del programa**.
- Escribe (println) en la pantalla estándar (System.out) la frase "Hola Mundo!".



## 5. VARIABLES

### 5.1 Qué son y cómo se declaran las variables

En nuestro primer ejemplo escribíamos un texto en pantalla, pero este texto estaba prefijado dentro de nuestro programa.

Esto no es lo habitual: **normalmente los datos que maneje nuestro programa serán el resultado de alguna operación matemática o de cualquier otro tipo, a partir de datos introducidos por el usuario, leídos de un fichero, obtenidos de Internet... Por eso, necesitaremos un espacio en el que ir almacenando valores temporales y resultados de las operaciones.**

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"**.

Por ejemplo, **si queremos que el usuario introduzca dos números y el ordenador calcule la suma de ambos números y la muestre en pantalla**, necesitaríamos el espacio para almacenar al menos esos dos números iniciales. No sería imprescindible reservar espacio también para la suma, porque podemos mostrarla en pantalla nada más calcularla, sin almacenarla previamente en ningún sitio). Los pasos a dar serían los siguientes:

1. Pedir al usuario que introduzca un número.
2. Almacenar ese valor que introduzca el usuario (por ejemplo, en un espacio de memoria al que podríamos asignar el nombre "primerNumero").
3. Pedir al usuario que introduzca otro número.
4. Almacenar el nuevo valor (por ejemplo, en otro espacio de memoria llamado "segundoNumero").
5. Mostrar en pantalla el resultado de sumar "primerNumero" y "segundoNumero".

Pues bien, en este programa estaríamos empleando dos variables llamadas "primerNumero" y "segundoNumero". Cada una de ellas la usaría para acceder a un espacio de memoria, que será capaz de almacenar un número.

Para no desperdiciar la memoria de nuestro ordenador, el espacio de memoria que hace falta "reservar" será distinto según lo grande que pueda llegar a ser dicho número (la cantidad de cifras), o según la precisión que necesitemos para ese número (cantidad de decimales). Por eso, tenemos disponibles diferentes "tipos de variables".

Por ejemplo, si vamos a manejar números sin decimales ("números enteros") de como máximo 9 cifras, nos interesaría el tipo llamado "int" (abreviatura de "integer", "entero" en inglés). Este tipo de datos consume un espacio de memoria de 4 bytes. Si no necesitamos guardar datos tan grandes (por ejemplo, si nuestros datos va a ser números inferiores a 1.000), podemos emplear el tipo de datos llamado "short" (entero "corto"), que ocupa la mitad de espacio.

Con eso, vamos a ver un programa que sume dos números enteros (de no más de 9 cifras) prefijados y muestre en pantalla el resultado:

```
/* -----* * Introducción a Java - Ejemplo * *
Archivo: Suma.java * (Suma dos números enteros)-----*/
class Suma {
public static void main( String args[] ) {
int primerNumero = 56; // Dos enteros con valores prefijados
int segundoNumero = 23;
System.out.println( "La suma es:" ); // Muestro un mensaje de aviso
System.out.println(primerNumero+segundoNumero ); // y el resultado de la operación
}
}
```

Como se ve, la forma de "declarar" una variable es detallando primero el tipo de datos que podrá almacenar ("int", por ahora) y después el nombre que daremos la variable. Además, se puede indicar un valor inicial.

Hay una importante diferencia entre las dos órdenes "println": la primera contiene comillas, para indicar que ese texto debe aparecer "tal cual", mientras que la segunda no contiene comillas, por lo que no se escribe el texto " primerNumero+segundoNumero", sino que se intenta calcular el valor de esa expresión (en este caso, la suma de los valores que en ese momento almacenan las variables primerNumero y segundoNumero,  $56+23 = 89$ ).

Podríamos pensar en mejorar el programa anterior para que los números a sumar no estén prefijados, sino que se pida al usuario... pero eso no es trivial. El lenguaje Java prevé que quizá se esté utilizando desde un equipo que no sea un ordenador convencional, y que quizá no tenga un teclado conectado, así que deberemos hacer ciertas comprobaciones de errores que todavía están fuera de nuestro alcance. Por eso, vamos a aplazar un poco eso de pedir datos al usuario.

Los tipos de datos numéricos disponibles en Java son los siguientes:

Nombre	¿Admite decimales?	Valor mín.	Valor max.	Precisión	Ocupa
<b>byte</b>	no	-128	127	-	1 byte
<b>short</b>	no	-32768	32767	-	2 bytes
<b>int</b>	no	- 2.147.483.648	2.147.483.647	-	4 bytes
<b>long</b>	no	- 9.223.372.036 .854.775.808	9.223.372.036 .854.775.807	-	8 bytes
<b>float</b>	si	1.401298E-45	3.402823E38	6-7 cifras	4 bytes
<b>double</b>	si	4.9406564584 1247E-324	1.7976931348 6232E308	14-15 cifras	8 bytes

La forma de "declarar" variables de todos estos tipos es, como ya habíamos comentado, detallando primero el tipo de datos y después el nombre que daremos la variable, así:

```
int numeroEntero; // La variable numeroEntero será un número de tipo "int"
short distancia; // La variable distancia guardará números "short"
Long gastos; // La variable gastos es de tipo "Long"
byte edad; // Un entero de valores "pequeños"
float porcentaje; // Con decimales, unas 6 cifras de precisión
```

```
double numPrecision; // Con decimales y precisión de unas 14 cifras
```

Se pueden declarar varias variables del mismo tipo "**a la vez**":

```
int primerNumero, segundoNumero; // Dos enteros
```

(Eso sí, algunos autores recomiendan que no se haga: nuestro programa será más legible si escribimos en cada línea una variable distinta, y todas ellas precedidas por su tipo).

También se puede dar un valor a las variables a la vez que se declaran:

```
int a = 5; // "a" es un entero, e inicialmente vale 5  
short b=-1, c, d=4; // "b" vale -1, "c" vale 0, "d" vale 4
```

Los **nombres** de variables pueden contener **letras y números** (pero **no pueden comenzar con un número**) y algún otro **símbolo**, como el de **subrayado**, pero no podrán contener otros muchos símbolos, como los de las distintas operaciones matemáticas posibles (+,-,\*,/), ni llaves o paréntesis, ni vocales acentuadas (á,é,í,ó...), ni eñes...

Tenemos otros dos tipos básicos de variables, que no son para datos numéricos, y que usaremos más adelante:

- **char**. Será una letra del alfabeto, o un dígito numérico, o un símbolo de puntuación. Ocupa 2 bytes. 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 byte.

Estos ocho tipos de datos son lo que se conoce como "**tipos de datos primitivos**" (porque forman parte del lenguaje estándar, y a partir de ellos podremos crear otros más complejos).

## Constantes

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

```
final float PI = 3.14;
```

La palabra clave final es la que hace que PI sea un valor que no pueda ser modificado.

### 5.2 Operaciones matemáticas básicas.

Hay varias operaciones matemáticas que son frecuentes. Veremos cómo expresarlas en Java, y también veremos otras operaciones "menos frecuentes". Las que usaremos con más frecuencia son:

Operación	Símbolo
Suma	+
Resta	-
Multiplicación	*
División	/
Resto de la División	%

Hemos visto un ejemplo de cómo calcular la suma de dos números; las otras operaciones se emplearían de forma similar.

### 5.3 Incremento 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 entre un cierto número, `*=` para multiplicarla por un número, y así sucesivamente. 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;  
a+=1;  
a++;
```

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++;
```

da como resultado `c = 6` y `b = 5`, porque se asigna el valor a "b" antes de incrementar "c", mientras que

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

da como resultado  $c = 6$  y  $b = 6$  (se asigna el valor a "b" después de incrementar "c").

Por eso, para evitar efectos colaterales no esperados, es mejor no incrementar una variable a la vez que se asigna su valor a otra, sino hacerlo en dos pasos.

#### 5.4 Operadores relacionales

También podremos comprobar si entre dos números (o entre dos variables) existe una cierta relación del tipo "¿es a mayor que b?" o "¿tiene a el mismo valor que b?". Los operadores que utilizaremos para ello son:

Operación	Símbolo
Mayor que	>
Mayor o igual que	>=
Menor que	<
Menor o igual que	<=
Igual que	== (dos símbolos de igual)
Distinto de	!=

Así, por ejemplo, para ver si el valor de una variable "b" es distinto de 5, escribiríamos algo parecido (veremos la sintaxis correcta un poco más adelante) a

SI  $b \neq 5$  ENTONCES ...

o para ver si la variable "a" vale 70, sería algo como (nuevamente, veremos la sintaxis correcta un poco más adelante)

SI  $a == 70$  ENTONCES ...

Es muy importante recordar esta diferencia: para asignar un valor a una variable se emplea un único signo de igual, mientras que para comparar si una variable es igual a otra (o a un cierto valor) se emplean dos signos de igual.

### 5.5 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:

Operación	Símbolo
Y	&&
O	
No	!

Por ejemplo, la forma de decir "si a vale 3 y b es mayor que 5, o bien a vale 7 y b no es menor que 4" en una sintaxis parecida a la de Java (aunque todavía no es la correcta) sería:

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

### Precedencia de los operadores

Al igual que en las matemáticas en la programación, muchas veces es importante en orden de precedencia que tienen los operadores, para así pensar correctamente el algoritmo, por ello en la



siguiente tabla se muestra la precedencia separada en grupos de más a menos nivel de prioridad.

```
grupo 0: ( )
grupo 1: ++ -- +(unario) -(unario) !
grupo 2: * / %
grupo 3: + -
grupo 5: > >= < <=
grupo 6: == !=
grupo 7: &
grupo 8: ^
grupo 9: |
grupo 10: &&
grupo 11: ||
grupo 12: ?: (operador ternario)
grupo 13: = op= (op es uno de: +,-,*,/,%,&,|,^)
```

## 6. COMPROBACIÓN DE CONDICIONES

### 6.1 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 los pasos que queremos dar si se cumple la condición, así :

```
if (condición) {
    sentencias
}
```

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 haya que hacer varias cosas:

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

## 6.2 else

Una primera mejora es indicar **qué hacer en caso de que no se cumpla la condición**. Sería algo parecido a

```
SI condición_a_comprobar ENTONCES  
    pasos_a_dar  
EN_CASO_CONTRARIO  
    pasos_alternativos
```

que en Java escribiríamos así:

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

Por ejemplo,

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

### 6.3 switch

Si queremos comprobar varias condiciones, podemos utilizar varios if - else - if - else - if encadenados, pero tenemos una forma más elegante en Java de elegir entre distintos valores posibles que tome una cierta expresión. Su formato es éste:

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

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 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;  
}
```

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;  
}
```

Podemos conseguir que se den los mismos pasos en varios casos, simplemente eliminando la orden "break" de algunos de ellos. Así, un ejemplo algo más completo podría ser:

```
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" );
valorTemporal = 10;
System.out.println( "Operaciones auxiliares completadas" );
break;
default: System.out.println( "El valor de x no estaba entre 1 y 6" ); break;
}
```

#### 6.4 El operador condicional

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 condicional (?)":

```
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.

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;
```

## 7. PARTES DEL PROGRAMA QUE SE REPITEN

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.

### 7.1 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)
    sentencia;
```

Es decir, la sintaxis es similar a la de "if", con la diferencia de que aquella orden realizaba la sentencia indicada una vez como máximo (si se cumplía la condición), pero "while" puede repetir la sentencia más de una vez (mientras la condición sea cierta). Al igual que ocurría con "if", podemos realizar varias sentencias seguidas (dar "más de un paso") si las encerramos entre llaves:

```
x = 20;
while (x > 10){
    System.out.println( "Aún no se ha alcanzado el valor límite!");
    x--;
}
```

### 7.2 do- while

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

```
do{
    sentencia;
} while(condición);
```

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 (un único "while" antes de las sentencias), porque con "while", si la condición era falsa desde un principio, los pasos que se indican a continuación de "while" no llegaban a darse ni una sola vez.

Un ejemplo típico de esta construcción "do..while" es cuando queremos que el usuario introduzca una contraseña que le permitirá acceder a una cierta información:

```
do{  
    System.out.println( "Introduzca su clave de acceso: ");  
    claveIntentada = LeerDatosUsuario();  
  
} while(claveIntentada != claveCorrecta);
```

En este ejemplo hemos supuesto que existe algo llamado "LeerDatosUsuario", para que resulte legible. Pero realmente la situación no es tan sencilla: no existe ese "LeerDatosUsuario", ni sabemos todavía cómo leer información del teclado cuando trabajamos en modo texto (porque supondría hablar de excepciones y de otros conceptos que todavía son demasiado avanzados para nosotros), ni sabemos crear programas "de ventantas" en los que podamos utilizar una casilla de introducción de textos. Así que de momento nos creeremos que algo parecido a lo que hemos escrito podrá llegar a funcionar... aunque todavía no lo hace.

### 7.3 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. Su formato es

```
for ( valor_inicial ; condicion_continuacion ; incremento ) {  
    sentencias
```

```
}
```

Es decir, indicamos entre paréntesis, y separadas por puntos y coma, tres órdenes:

- La primera orden dará el valor inicial a una variable que sirva de control.
- La segunda orden será la condición que se debe cumplir mientras que se repitan las sentencias.
- La tercera orden será la que se encargue de aumentar -o disminuir- el valor de la variable, para que cada vez quede un paso menos por dar.

Esto se verá mejor con un ejemplo. Podríamos repetir 10 veces un bloque de órdenes haciendo:

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

(inicialmente i vale 1, hay que repetir mientras sea menor o igual que 10, y en cada paso hay que aumentar su valor una unidad),

O bien podríamos contar descendiendo desde el 10 hasta el 2, con saltos de 2 unidades en 2 unidades, así:

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

Nota: 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:

```
j = 10;  
while ( j > 0 ){  
    System.out.println( 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 sin fin los mismos pasos.



## 7.4 break y continue

Se puede modificar un poco el comportamiento de estos 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. Como ejemplo:

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

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"). Como ejemplo:

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

## 8. ARRAYS Y CADENAS DE TEXTO

Hemos visto cómo manejar tipos de datos básicos: varios tipos de datos numéricos, letras (char) y valores verdadero/falso (boolean). Pero para poder empezar a aplicar nuestros conocimientos en ejemplos medianamente complicados, nos interesa ver al menos un par de tipos de datos más:

- Es muy habitual tener que manejar "bloques" de letras, que darán lugar a palabras o frases. Estas serán las "cadenas de texto" o "strings".
- También es frecuente tener que manejar "bloques" de números. Es algo que sonará familiar a quien haya estudiado estadística o álgebra matricial, por ejemplo. Para eso utilizaremos los "arrays" (palabra que algunos autores traducen por "arreglos", y que, en ciertos contextos, equivalen a "matrices" o a "vectores").

Veamos una introducción a ambos tipos de datos.

### 8.1 Los arrays

Imaginemos que tenemos que hallar el promedio de 10 números que introduzca el usuario (o realizar cualquier otra operación con ellos). Parece evidente que tiene que haber una solución más cómoda que definir 10 variables distintas y escribir 10 veces las instrucciones de avisar al usuario, leer los datos que teclee, y almacenar esos datos. Si necesitamos manejar 100, 1.000 o 10.000 datos, resulta todavía más claro que no es eficiente utilizar una variable para cada uno de esos datos.

Por eso se emplean los arrays (o arreglos). **Un array es una variable que puede contener varios datos del mismo tipo.** Para acceder a cada uno de esos datos emplearemos corchetes. Por ejemplo, si definimos una variable llamada "m" que contenga 10 números enteros, accederemos al primero de estos números como m[0], el último como m[9] y el quinto como m[4] (se empieza a numerar a desde 0 y se termina en n-1). Veamos un ejemplo que halla la media de cinco números (con decimales, "double"):

```
// Array1.java
// Aplicación de ejemplo con Arrays
// Introducción a Java,
```

```
class Array1 {  
    public static void main( String args[] ) {  
        double a[] = { 10, 23.5, 15, 7, 8.9 };  
        double total = 0;  
        int i;  
        for (i=0; i<5; i++)  
            total += a[i];  
  
        System.out.println( "La media es:" );  
        System.out.println( total / 5 );  
    }  
}
```

Para definir la variable podemos usar **dos formatos**: "double a[]" (que es la sintaxis habitual en C y C++) o bien "**double[] a**" ( que es la sintaxis recomendada en Java, y posiblemente es una forma más "razonable" de escribir "la variable a es un array de doubles").

Lo habitual no será conocer los valores en el momento de teclear el programa, como hemos hecho esta vez. Será mucho más frecuente que los datos los teclee el usuario o bien que los leamos de algún fichero, los calculemos, etc. En este caso, tendremos que reservar el espacio, y los valores los almacenaremos a medida que vayamos conociéndolos. Para ello, primero **declararemos** que vamos a utilizar un array, así:

**double[]** datos;

y después **reservaremos** espacio (por ejemplo, para 1.000 datos) con

datos = **new double** [1000];

Estos dos pasos se pueden dar en uno solo, así:

**double[]** datos = **new double** [1000]; y daríamos los **valores** de una forma similar a la que hemos visto en el ejemplo anterior:

datos[25] = 100 ;

datos[0] = i\*5 ;

datos[j+1] = (j+5)\*2;

Vamos a ver un ejemplo algo más completo, con tres arrays de números enteros, llamados a, b y c. A uno de ellos (a) le daremos valores al definirlo, otro lo definiremos en dos pasos (b) y le daremos fijos, y el otro lo definiremos en un paso y le daremos valores calculados a partir de ciertas operaciones:

```
// Array2.java
// Aplicación de ejemplo con Arrays
// Introducción a Java

class Array2 {
    public static void main( String args[] ) {

        int i; // Para repetir con bucles "for"

        // ----- Primer array de ejemplo
        int[] a = { 10, 12345, -15, 0, 7 };
        System.out.println( "Los valores de a son:" );
        for (i=0; i<5; i++)
            System.out.println( a[i] );

        // ----- Segundo array de ejemplo
        int[] b;
        b = new int [3];
        b[0] = 15;
        b[1] = 132;
        b[2] = -1;
        System.out.println( "Los valores de b son:" );
        for (i=0; i<3; i++)
            System.out.println( b[i] );

        // ----- Tercer array de ejemplo
        int j = 4;
        int[] c = new int[j];
```

```
for (i=0; i<j; i++)
    c[i] = (i+1)*(i+1);
System.out.println( "Los valores de c son:" );
for (i=0; i<j; i++)
    System.out.println( c[i] );
} }
```

## 8.2 Las cadenas de texto

Una cadena de texto (en inglés, "string") es un **bloque de letras, que usaremos para poder almacenar palabras y frases**. En algunos lenguajes, podríamos utilizar un "array" de "chars" para este fin, pero en Java no es necesario, porque tenemos un tipo "cadena" específico ya incorporado en el lenguaje.

Realmente en Java hay dos "variantes" de las cadenas de texto: existe una clase llamada **"String"** y otra clase llamada **"StringBuffer"**. Un **"String"** será una **cadena de caracteres constante**, que no se podrá modificar (podremos leer su valor, extraer parte de él, etc.; para cualquier modificación, realmente Java creará una nueva cadena), mientras que un **"StringBuffer"** **se podrá modificar "con más facilidad"** (podremos insertar letras, dar la vuelta a su contenido, etc) a cambio de ser ligeramente menos eficiente (más lento).

Vamos a ver las principales posibilidades de cada uno de estos dos tipos de cadena de texto y luego lo aplicaremos en un ejemplo.

Podemos "concatenar" cadenas (juntar dos cadenas para dar lugar a una nueva) con el signo +, igual que sumamos números. Por otra parte, los métodos de la clase String (las "operaciones con nombre" que podemos aplicar a una cadena) son:

Método	Cometido
length()	Devuelve la longitud (número de caracteres) de la cadena
charAt(int pos)	Devuelve el carácter que hay en cierta posición

toLowerCase()	Devuelve la cadena convertida a minúsculas
toUpperCase()	Devuelve la cadena convertida a mayúsculas
substring(int desde, int cuantos)	Devuelve una subcadena: varias letras a partir de una posición dada
replace(char antiguo, char nuevo)	Devuelve una cadena con un carácter reemplazado por otro
trim()	Devuelve una cadena sin espacios blanco iniciales ni finales
startsWith(String subcadena)	Indica si la cadena empieza con una cierta subcadena
endsWith(String subcadena)	Indica si una cadena termina con una cierta subcadena
indexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el principio a partir de una posición opcional)
lastIndexOf(String subcadena, [int desde])	Indica la posición en que se encuentra una cierta subcadena (buscando desde el final a partir de una posición opcional)
valueOf(objeto)	Devuelve un String que es la representación como texto del objeto que se le indique (número, boolean, etc.)
concat(String cadena)	Devuelve la cadena con otra añadida al final
equals(String cadena)	Mira si dos cadenas 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)

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:

```
// Strings1.java
// Aplicación de ejemplo con Strings
// Introducción a Java

class Strings1 {
    public static void main( String args[] ) {

        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) );
System.out.println( "Comparamos texto1 y texto2: "
+ texto1.compareTo(texto2) );

} }
```

El resultado de este programa 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

Comparamos texto1 y texto2: -8



## 9. DATOS INTRODUCIDOS POR EL USUARIO

### 9.1 Cómo leer datos desde consola con la clase Scanner

A partir de la versión 5 de Java, tenemos la posibilidad de acceder a la entrada de teclado con la clase Scanner. En su uso más sencillo, le indicamos qué flujo de datos debe analizar (ya sea un fichero o la entrada estándar del sistema, "System.in"), y vamos obteniendo el siguiente dato con ".next" (el siguiente hasta llegar a un espacio en blanco):

```
//  
// Pedir datos al usuario de forma mas simple,  
// palabra por palabra (Java 5 o superior)  
//  
import java.io.*; import java.util.Scanner;  
  
class Scanner1 {  
    public static void main( String args[] ) throws IOException {  
        String nombre;  
  
        System.out.print( "Introduzca su nombre (una palabra): " );  
  
        Scanner entrada=new Scanner(System.in);  
        nombre = entrada.next();  
  
        System.out.println( "Hola, " + nombre );  
    }  
}
```

No sólo podemos leer cadenas de texto. Si lo siguiente que queremos leer es un número, podemos usar ".nextInt", ".nextFloat", ".nextDouble"... Y si queremos obtener más de un dato, podemos repetir con ".hasNext" ("tiene siguiente"), que nos devolverá verdadero o falso.

Típicamente se usaría como parte de un bucle "while".

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

### 9.2 Limpiar el buffer de entrada de la clase Scanner

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:

```
n = teclado.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 = sc.nextLine(); //leer un String
```

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 JavaApplication {  
    public static void main(String[ ] args) {  
        Scanner entrada = new Scanner(System.in);  
        String nombre;
```

```
        double radio;
        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(); //leemos el String después del
entero

        System.out.println("Hola " + nombre + "!!!");
    }
}
```

¿Cómo lo podemos solucionar?

```
public class JavaApplication {

    public static void main(String[] args) {

        Scanner entrada = new Scanner(System.in);
        String nombre;
        double radio;
        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 el String después del
entero
    }
}
```

```
        System.out.println("Hola " + nombre + "!!!");  
    }  
}
```

## 10. PARSE

La conversión de Strings a valores numéricos 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.

Además, estas clases contienen métodos que nos permiten manejar dichos objetos.

Normalmente, la envoltura posee el mismo nombre que el tipo de dato primitivo, aunque con la primera letra en mayúscula.

En la siguiente tabla vemos todas las posibilidades:

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.valueOf(aString.booleanValue())

Cada una de estas clases envoltura posee un método mediante el cual podemos convertir el String al tipo primitivo específico. Solo debemos usar el método desde la clase envoltura apropiada:

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

