IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones multiplataforma

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

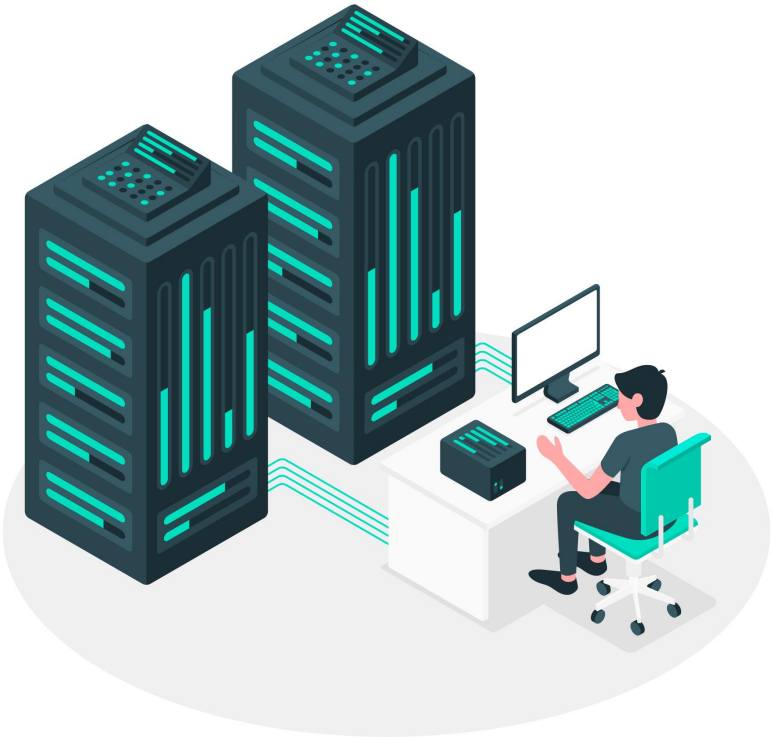
GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



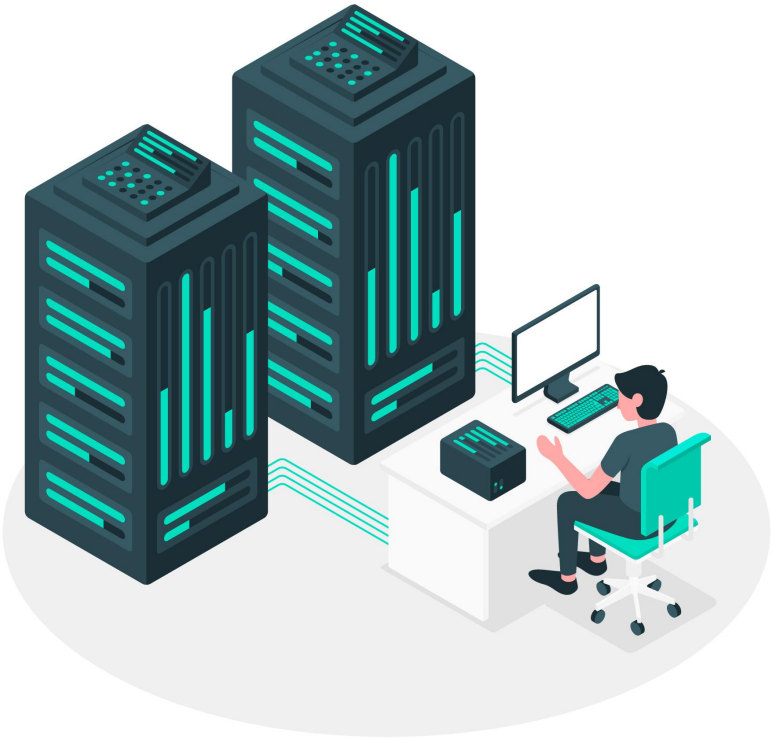
Material elaborado por:

Anna Sanchis Perales

Lionel Tarazón

Revisado y editado por:

Edu Torregrosa Llácer



Datos profesor

Edu Torregrosa Llácer

eduardotorregrosa@ieslluissimarro.org

Web del módulo: https://aules.edu.gva.es/fp/login/index.php

Introducción a JAVA

1. Introducción.
2. JDK y Visual Studio Code.
3. Mi primer algoritmo en JAVA.
4. Variables.
5. Condicionales
6. Bucles
7. Arrays y cadenas
8. Introducción de datos
9. Conversiones de tipos
10. Depuración

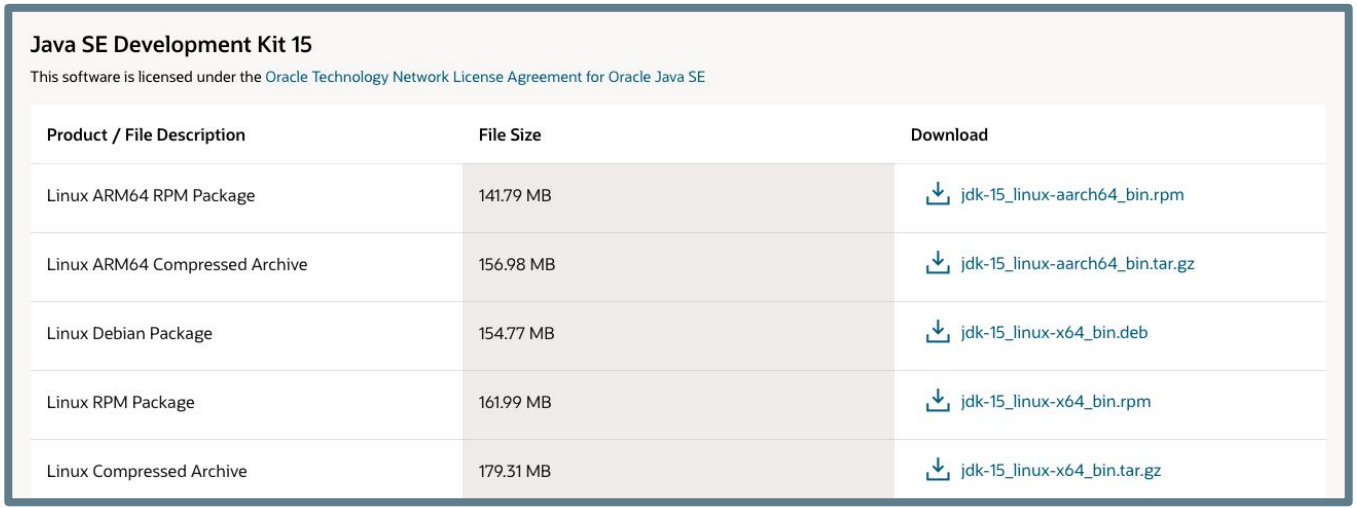
Programas en JAVA

- **¿Qué hace falta para usar un programa creado en Java?**
  - → Máquina virtual JAVA (**JVM**)
  - → Java Runtime Environment (**JRE**)
- **¿Qué hace falta para crear un programa en Java?**
  - → Java Development Kit (**JDK**)
  - → Entorno de desarrollo (Eclipse, NetBeans, VS Code)

Java Development Kit (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



Java SE Development Kit 15

This software is licensed under the Oracle Technology Network License Agreement for Oracle Java SE

Product / File Description	File Size	Download
Linux ARM64 RPM Package	141.79 MB	jdk-15_linux-aarch64_bin.rpm
Linux ARM64 Compressed Archive	156.98 MB	jdk-15_linux-aarch64_bin.tar.gz
Linux Debian Package	154.77 MB	jdk-15_linux-x64_bin.deb
Linux RPM Package	161.99 MB	jdk-15_linux-x64_bin.rpm
Linux Compressed Archive	179.31 MB	jdk-15_linux-x64_bin.tar.gz

Visual Studio Code

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

Visual Studio Code

- Accederemos a su web para iniciar la **descarga**: **https://code.visualstudio.com/**
- Tras ello procederemos con su instalación. Una vez instalado procederemos a personalizarlo para que soporte el lenguaje de programación Java. En el apartado "**Customize**" instalamos el soporte para Java.

Mi primer programa en JAVA - Hola mundo!!

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

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

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

Mi primer programa en VS Code

1. En primer lugar, deberemos crear una carpeta donde guardaremos el código Java. Una buena estructura sería crear una carpeta **CodigoJava** y dentro de ella una carpeta por cada tema, en este caso crearemos la **Tema3**.
2. Tras ello abriremos la carpeta con la opción Open folder…
3. 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
4. En el fichero que hemos creado copiaremos el código Java de Hola Mundo!

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

Mi primer programa en VS Code

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

   Terminal > New Terminal

6. Procedemos a compilar con la siguiente orden:

```
javac HolaMundo.java
```

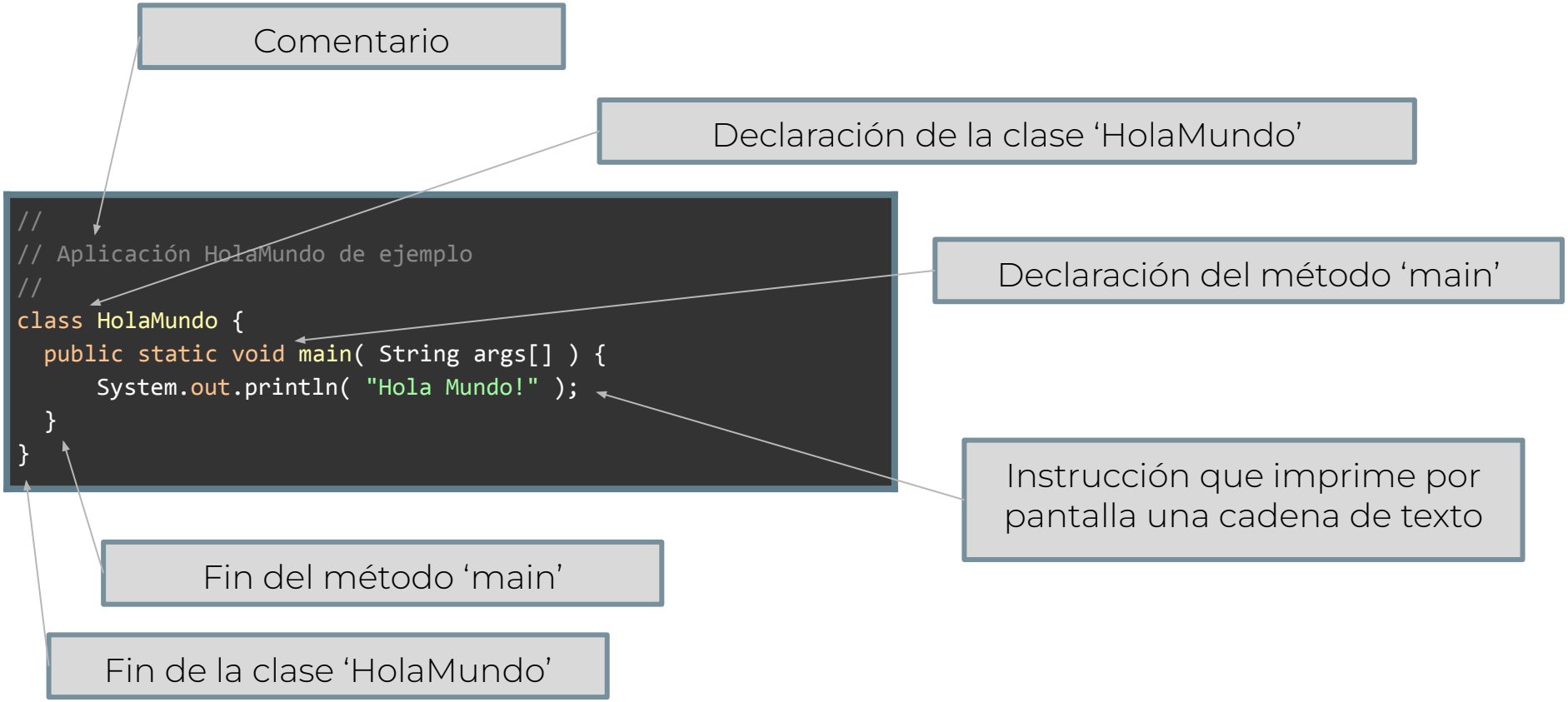
7. Y a ejecutar:

```
java HolaMundo
```

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

```
Hola Mundo!
```

Entendiendo mi primer programa en JAVA




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

Variables

Normalmente los datos que manejan los programa son el resultado de alguna operación, ya sea matemática o de cualquier otro tipo.

Estas operaciones se realizan a partir de datos introducidos por el usuario, leídos de un fichero, obtenidos de Internet... Por eso, necesitaremos un espacio donde podamos almacenar los valores temporales y los 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**".

Para no desperdiciar la memoria de nuestro ordenador, el espacio de memoria que hace falta "**reservar**" será distinto según el "**tipo de datos**" que queramos almacenar(enteros, reales, caracteres, cadenas, etc).

Tipos de variables en JAVA

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

Nombre	¿Admite decimales?	Valor mín	Valor máx	Precisión	Ocupa
byte	no	-128	127	-	1 byte
short	no	-32768	32767	-	2 bytes
int	no	-2.147.483.648	2.147.483.647	-	4 bytes
long	no	-9.223.372.036 .854.775.808	9.223.372.036. 854.775.807	-	8 bytes
float	si	1.401298E-45	3.402823E38	6-7 cifras	4 bytes
double	si	4.940656458412 47E-324	1.797693134862 32E308	14-15 cifras	8 bytes

Tipos de variables en JAVA

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 "false" (false). Ocupa 1 byte.

Declarar variables en JAVA

La forma de "declarar" una variable es detallando primero el tipo de datos que podrá almacenar y después el nombre que daremos la variable. Además, se puede indicar un valor inicial.

```
//Ejemplo de declaraciones de variables
class Suma {
public static void main( String args[] ) {
int primerNumero = 56;  // declaramos variable primerNumero y asignamos valor 56
int segundoNumero = 23; // declaramos variable segundoNumero y asignamos valor 23
System.out.println( "La suma es:" ); // Muestro un mensaje
System.out.println(primerNumero+segundoNumero ); // y el resultado de la operación
}//cerramos main
}//cerramos clase
```

Declarar variables en JAVA

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).

```
//Ejemplo de declaraciones de variables
class Suma {
public static void main( String args[] ) {
int primerNumero = 56;  // declaramos variable primerNumero y asignamos valor 56
int segundoNumero = 23; // declaramos variable segundoNumero y asignamos valor 23
System.out.println( "La suma es:" ); // Muestro un mensaje
System.out.println(primerNumero+segundoNumero ); // y el resultado de la operación
}//cerramos main
}//cerramos clase
```

Declarar variables en JAVA

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

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

```
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...

Declarar variables en JAVA

Identificador de la variable: es el nombre que damos a la variable

Deben seguir las siguientes reglas:

- Debe comenzar por letra, subrayado (_) o el carácter $. No pueden comenzar por un número.
- Después del primer carácter si que se pueden utilizar números.
- No pueden utilizarse espacios en blanco ni símbolos de operadores.
- No pueden coincidir con ninguna palabra reservada.
- El % no está permitido, si el $ y la ç. También los acentos y la ñ, pero no se recomienda utilizarlos.

Declarar variables en JAVA

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

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

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.

Cadenas de texto en JAVA

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.

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:

Cadenas de texto en JAVA

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

Cadenas de texto en JAVA

Método	Cometido
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 cadena son iguales (lo mismo que ==)
equalsIgnoreCase(String cadena)	Mira si dos cadena son iguales, pero despreciando las diferencias entre mayúsculas y minúsculas
compareTo(String cadena2)	Compara una cadena con la otra (devuelve 0 si son iguales, negativo si la cadena es menor que cadena2 y positivo si es mayor

Secuencias de escape

\n	Salto de línea. Sitúa el cursor al principio de la línea siguiente
\b	Retroceso. Mueve el cursor un carácter atrás en la línea actual.
\t	Tabulador horizontal. Mueve el cursor hacia adelante una distancia determinada por el tabulador.
\r	Ir al principio de la línea. Mueve el cursor al principio de la línea actual.
\f	Nueva página. Mueve el cursor al principio de la siguiente página.
\”	Comillas. Permite mostrar por pantalla el carácter comillas dobles.
\’	Comilla simple. Permite mostrar por pantalla el carácter comilla simple.
****	Barra inversa.
\unicode	Carácter Unicode. d representa un dígito hexadecimal del caracter Unicode.

Cadenas de texto en JAVA

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

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

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

Cadenas de texto en JAVA

```
// 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) );
   }
}
```

Cadenas de texto en JAVA

El resultado tras ejecutar el programa anterior sería el siguiente:

La primera cadena de texto es :

Hola

Concatenamos las dos: HolaPrueba

Concatenamos varios: Hola5 23.5

La longitud de la segunda es: 6

La segunda letra de texto2 es: r

La cadena texto2 en mayúsculas: PRUEBA

Tres letras desde la posición 1: ru

Comparamos texto1 y texto2: -8

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	%

Incrementos y asignaciones abreviadas

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

Al igual que tenemos el operador **+=** para aumentar el valor de una variable, tenemos **-=** para disminuirlo, **/=** para dividirla 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++;
```

Incrementos y decremento

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

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

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

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

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

Incrementos y decremento

Sin embargo:

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

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

Ejemplos incrementos y decremento

```
int a=1, b=2, c=3;
b = ++c; //b vale 4 y c vale 4
a += b++; //a vale 5 y b vale 5
a = a+++a; //a vale 11
a -= b--; //a vale 6 y b vale 4
c = a++ - ++b; //c vale 1, a vale 7 y b vale 5
c -= ++a; //c vale -7 y a vale 8
a -= ++c; //a vale 14 y c vale -6
a -= c++; //a vale 20 y c vale -5
a -= --c; // a vale 26 y c vale -6
```

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	!=

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 el 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: +,-,*,/,%,&,|,^)
```

Condiciones (IF)

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

```
SI condición_a_comprobar ENTONCES
    pasos_a_dar
```

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

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

Condiciones (IF)

Por ejemplo:

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

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

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

Condiciones (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
}
```

Condiciones (ELSE)

Por ejemplo:

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

Condiciones (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
}
```

Condiciones (SWITCH)

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

Un ejemplo sería:

```
 switch ( x * 10) {
   case 30: System.out.println( "El valor de x 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;
}
```

Condiciones (SWITCH)

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

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

Por ejemplo así:

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

Condiciones (SWITCH)

Podemos conseguir que 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;
}
```

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.

El operador condicional

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;
```

Bucles

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

BUCLE WHILE

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

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

Bucles

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

BUCLE DO-WHILE

En este caso, la condición se comprueba al final, lo que quiere decir que las "**sentencias**" **intermedias se realizarán al menos una vez**, cosa que no ocurría en la construcción anterior (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.

```
do{
      sentencia1;
      sentencia2;
       …
      sentenciaN;

} while (condición);
```

Bucles

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.

BUCLE FOR

```
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.

Bucles

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 );
```

Bucles

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

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

```
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.

Break

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( "Comienza 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.

Continue

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

Ejemplos Bucles

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

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

```
10
8
6
4
2
```

Ejemplos Bucles

Es muy importante tener claro el orden de ejecución de las intrucciones en un bucle FOR:

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

Orden	Instrucción	j
1	j=10	10
2	j>0	10
3	println	**10**
4	j-=2	8
5	j>0	8
6	println	**8**
7	j-=2	6
8	j>0	6
9	println	**6**
.....	.....	...
16	j-=2	0
17	j>0	0
18	FIN	

Ejemplos Bucles

A continuación se muestran un ejemplo con dos bucles anidados:

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

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

Ejemplos Bucles

A continuación se muestran un ejemplo con dos bucles anidados:

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

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

no entra, i>b

si entra, i=b

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

ENTRADA Y SALIDA DE DATOS

Salida por pantalla

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

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

```
System.out.print("A: "+a);
```

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

```
System.out.println("A: "+a);
```

Datos introducidos por el usuario

A partir de la versión 5 de Java, tenemos la posibilidad de acceder a la entrada de teclado con la **clase Scanner.**

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

```
// Pedir datos al usuario de forma 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 );
  }
}
```

Datos introducidos por el usuario

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

- ➔ nombre_variable_scanner.nextInt()
- ➔ nombre_variable_scanner.nextFloat()
- ➔ nombre_variable_scanner.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()) {
```

Datos introducidos por el usuario

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

Esto es, la instrucción:

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

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

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

```
System.out.print("Introduzca su nombre: ");
nombre = entrada.nextLine();  //leer un String
```

Datos introducidos por el usuario

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

```
import java.util.Scanner;
public class 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 + "!!!");
     }
}
```

Datos introducidos por el usuario

¿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 + "!!!");
      }
}
```

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.

Parse

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()

Ejemplo:

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

Preguntas