

Índice

Índice

Introducción a Java

JDK y JRE

Máquina Virtual de Java (JVM)

Tipos de datos

Variables

Reglas para los identificadores

Tipos primitivos

Literales

Conversiones explícitas

Conversiones implícitas

Constantes

Entrada y salida de datos

Salida por pantalla

Colores

Entrada por teclado

Cerrar teclado

Buffer de teclado

Introducción a Java

Java es un lenguaje de programación orientado a objetos, multiplataforma, robusto y fuertemente tipado. La plataforma Java proporciona, además de un lenguaje de programación, todo un conjunto de especificaciones, tecnologías y librerías de clases, mediante los cuales se pueden crear diferentes tipos de programas informáticos capaces de ser ejecutados en una amplia variedad de sistemas operativos.

Esta independencia, tanto del sistema operativo como del dispositivo, ha contribuido a expandir el lenguaje, y a que numerosos fabricantes de software hayan apostado por el, y que a día de hoy siga siendo la principal opción para numerosas empresas y programadores a la hora realizar sus desarrollos.

JDK y JRE

{{Próximamente}}

Máquina Virtual de Java (JVM)

{{Próximamente}}

Tipos de datos

Java es un lenguaje fuertemente tipado, y se ha de especificar los tipos de datos en la creación de variables, objetos, atributos, valores devueltos por los métodos y parámetros recibidos por los métodos.

Variables

En programación, una variable está formada por un espacio en memoria y un nombre simbólico, llamado **identificador**, que queda asociado a dicho espacio. En ese espacio se podrá almacenar un **valor**. Usaremos el identificador para hacer referencia al valor almacenado en memoria. El valor de la variable puede cambiar durante la ejecución del programa.

Reglas para los identificadores

- Deben empezar por una letra y en minúscula. En Java se usa la notación camelCase.
- Pueden contener números, pero no podrán empezar por un número.
- No pueden usarse tildes, ni signos de puntuación, ni caracteres especiales.
- No podrán usarse palabras reservadas para el lenguaje.
- Identificadores válidos serían: `contador`, `contador3`, `c`, `x2`, `contadorVocales`.
- Identificadores no válidos serían: `1contador`, `123`, `c%`, `x-y`, `?min`, `año`, `spidermán`.
(Realmente si funcionaría la ñ y las tildes, pero nos provocarían muchos errores futuros, así que mejor tratarlos como no válidos.)

Para **definir una variable** (también llamado declarar) de un tipo de dato concreto, se usará la sintaxis siguiente:

```
1 <tipoDeDato> identificador;
```

Para poder usar una variable (operar con ella, mostrar su valor, etc), Java nos obliga a darle un valor inicial. Esto se llama **inicializar**. Para asignar un valor a una variable ya declarada, usaremos el operador de asignación `=`.

```
1 identificador = valor;
```

También se puede hacer la **declaración e inicialización** en una misma sentencia:

```
1 <tipoDeDato> identificador = valor;
```

Veamos los primeros tipos de datos en Java, antes de poder definir una variable.

Tipos primitivos

También llamados atómicos, puesto que son las unidades más pequeñas con las que podemos trabajar en Java. Los objetos estarán formados por otros objetos o bien por tipos primitivos. Los tipos primitivos en Java son:

Nombre	Declaración	Descripción	Memoria	Rango
Booleano	<code>boolean</code>	Define una variable que puede tomar dos posibles valores. <code>true</code> o <code>false</code> .	1 bit	<code>true</code> o <code>false</code>

Nombre	Declaración	Descripción	Memoria	Rango
Byte	<code>byte</code>	Define una variable entera pequeña.	1 byte	[-128 ... 127]
Short	<code>short</code>	Define una variable entera corta.	2 bytes	[-32.768 ... 32.767]
Entero	<code>int</code>	Define una variable entera estándar. Suele ser la más usada para representar un número entero.	4 bytes	$[-2^{31} \dots 2^{31}-1]$
Entero largo	<code>long</code>	Define una variable entera con un mayor rango, pero ocupa el doble en memoria.	8 bytes	$[-2^{63} \dots 2^{63}-1]$
Decimal simple	<code>float</code>	Define una variable real estándar.	4 bytes	$[\pm 3,4 \cdot 10^{-38} \dots \pm 3,4 \cdot 10^{38}]$
Decimal doble	<code>double</code>	Define una variable real con doble precisión. Ocupa el doble que <code>float</code> en memoria.	8 bytes	$[\pm 1,7 \cdot 10^{-308} \dots \pm 1,7 \cdot 10^{308}]$
Carácter	<code>char</code>	Define una variable que podrá almacenar un único carácter o su código ASCII. Se delimita con comillas simples. Ejemplo: <code>'a'</code> .	2 bytes	[0 .. 65.535]
String ¹	<code>String</code>	Define una variable que podrá almacenar una cadena de caracteres. Se delimita con comillas dobles. Ejemplo: <code>"Hola Mundo"</code> .		

Nota importante¹: El tipo de datos `String` NO ES una variable de tipo primitivo. Ya se verá más adelante a fondo. La incluimos aquí porque podemos catalogarla como **básica** e incluirla “moralmente” entre los tipos primitivos, aunque técnicamente no lo sea.

Una vez que ya hemos visto las variables, sus tipos de datos, y la sintaxis para declarar e inicializar variables, veamos nuestras primeras líneas de código real en Java, dónde definimos una variable de cada tipo:

```

1 //Booleanas
2 boolean esCorrecto = true; //true o false
3
4 //Enteros
5 byte b = 127;           //entre -128 y 127
6 short s = 32767;        //entre -32768 y 32767
7 int i = 2147483647;      //tipo entero
8 long l = 2150000000L;    //tipo entero largo;
```

```

9
10 //Decimales
11 float f = (float) 7.5;      //tipo float (decimal con simple precisión)
12 double d = 9.573;          //tipo double (decimal con doble precisión)
13
14 //char
15 char letra1 = 'A';          //Los char se pueden tratar indistintamente como
    un carácter
16 char letra2 = 65;           //o como un número
17 char letra3 = 'B' + 1;      //Esto NO DARÍA B1, si no 'C'
18
19 //String
20 String curso = "Desarrollador/a Fullstack"; //No sería un primitivo, pero
    casi ;)

```

Literales

Se denominan **valores literales** a aquellos datos que son introducidos en el código del programa directamente en forma de constantes. Suelen usarse para asignar un valor inicial a una variable o como parte de una expresión aritmética o de otro tipo.

- `int a = 15;` -> **15** sería un literal de número.
- `String cadena = "Hola Mundo!";` -> **Hola Mundo** sería un literal de String.

En Java, los literales también tienen su tipado, y siguen las siguientes reglas:

- Los literales numéricos, sean del tipo que sean, no se delimitan con ningún carácter.
- Los literales de números enteros, que no se definan explícitamente de un tipo, serán tomados como tipo `int`.
- Los literales de números decimales, que no se definan explícitamente de un tipo, serán SIEMPRE definidos como tipo `double`.
- Los literales de cadenas, se delimitan por comillas dobles `" "`.
- Los literales de carácter, se delimitan por comillas simples `' '`, o bien por un número entero
- Los literales booleanos, serán `true` o `false`.

Los tipos de datos numéricos, siempre que no se pierda información, pueden “caber” en variables de tipo de dato de igual tipo o más grande, no necesitando ninguna conversión explícita.

Si visualizamos el literal como un cajón, y la variable como una cajonera, entenderemos que nunca podremos guardar un cajón en una cajonera más pequeña, pero si al revés. Es decir, no podremos guardar un valor `double` dentro de una variable `float`, pero si podremos guardar un valor `int` en una variable `double` o `float`.

```

1  int a = 15;    //El 15, es un entero, que se mete en una variable de tipo
    entero. Correcto.
2  float b = 3;   //El 3, es un entero, que se mete en una variable de tipo
    float. Correcto.
3  float c = 2.4; //El 2.4, es un DOUBLE, que se mete en una variable de tipo
    float. ERROR. No cabe.

```

Conversiones explícitas

Si queremos guardar un literal decimal, en una variable de tipo `float`, deberemos convertir explícitamente el valor a `float`, ya que un literal decimal SIEMPRE será de tipo `double`, y no cabría en un `float`.

```
1 float nota = (float) 8.3; //Así convertimos explícitamente el double a float (casting)
2 float temp = 25.8f;      //Así también, pero no se recomienda.
```

Para guardar un literal numérico en un `long`, que supere el rango de los enteros, deberemos convertir explícitamente también el número a `long`, ya que un número literal por defecto es de tipo `int`, y si este supera el rango permitido, dirá que nos estamos saliendo del rango. Para convertirlo explícitamente, usaremos la letra `L` al final del número para indicar que es `long`.

```
1 long largo = 3147000000L;
```

Conversiones implícitas

Son las conversiones producidas de forma automática por Java, sin que explícitamente indiquemos el tipo de dato resultante.

Veamos las más comunes:

Al dividir dos números de tipo entero, el resultado será de tipo entero y no de tipo decimal, sea cual sea la operación:

```
1 int a=10;
2 System.out.println(a/2); //El resultado es -> 5
3 System.out.println(a/3); //El resultado es -> 3. No da 3.333333
```

Si queremos realizar una división que el resultado de decimales, deberemos usar en la expresión algún tipo de dato con decimales, como `float` o `double`, ya que tipo de dato del resultado de una expresión, será el tipo de dato mayor involucrado en la operación.

```
1 int a=10;
2 System.out.println(a/2.0); //El resultado es -> 5.0
3 System.out.println(a/3.0); //El resultado es -> 3.3333333
```

En el ejemplo anterior, ambos resultados dan decimales, porque ya no estamos dividiendo un `int` entre un `int`, como antes, si no un `int` entre un `double` (recordar que un literal decimal SIEMPRE es `double`). Como resultado de la operación, el tipo de dato resultante será el mayor involucrado, para que no se pierda información en la operación.

Veamos otra posibilidad:

```
1 float division1 = 5 / 2;      //Daría 2, y se guardaría 2.0
2 float division2 = 5 / 2.0;    //Error
3 float division3 = 5 / 2f;     //Funcionaría
4 float division4 = 5 / (float) 2; //Funcionaría
```

- Línea 1: Dividimos `5 / 2`, ambos enteros, el resultado es entero, y al guardarse en un `float`, se guardaría `2.0`.
- Línea 2: Dividimos `5 / 2.0`, un `int` entre un `double`, el resultado es `double`, por lo que no podemos almacenarlo en un `float`.
- Línea 3: Dividimos `5 / 2f`. Un `int` entre un `float`, el resultado es `float`, y se puede almacenar perfectamente en un `float`.

Con los String también tenemos conversiones implícitas:

Para convertir de char a String:

```
1 char letra = 'Y';
2 String nif = "11223344" + letra; //Resultado -> "11223344Y"
```

Si concatenamos un `char` a un `String`, el resultado será `String`.

Para convertir números (da igual el tipo) a String:

```
1 String nombre = "Max Power";
2 float cantidad = 3000;
3 System.out.println("Hola " + nombre + ", te queremos " + cantidad);
4 //Sale por pantalla -> "Hola Max Power, te queremos 3000"
```

En la expresión anterior, empezamos concatenando `"Hola " + nombre + ", te queremos "`. Sumamos 3 valores de `String`, que al concatenarlos obtenemos `Hola Max Power, te queremos`. Pero después le concatenamos un número. Ahí se hace una conversión implícita. El número se convierte a `String`, pasando de `3000` a `"3000"`, y se concatena al resto del `String`.

MUY IMPORTANTE: Las expresiones, como en todos los lenguajes de programación, se resuelven de izquierda a derecha, teniendo máxima prioridad los paréntesis. Después las funciones, la multiplicación y división, y por último la suma y la resta.

Veamos como de importante es el orden en la resolución de una expresión:

```
1 String cadena1 = "1" + 2 + 3; //Resultado "123"
2 String cadena2 = 1 + 2 + "3"; //Resultado 1+2=3 -> 3 + "3" -> "33"
3 String cadena3 = 1 + (2 + "3"); //Resultado 2+"3"= "23" -> 1 + "23" -> "123"
4 String cadena4 = "1" + (2 + 3); //Resultado (2+3=5) -> "1" + 5 -> "15"
5 String cadena5 = 1 + 2 + 3; //Daría error, no puede meter un int en un String
```

Constantes

Una constante en Java es una variable que su valor no podrá ser modificado después de su inicialización. A todos los efectos funciona exactamente igual que una variable, con la diferencia que si intentamos asignarle un valor cualquiera, una vez que ya tenga su valor inicial, provocará un error de compilación y el propio IDE nos indicará el error.

Las constantes se declaran igual que las variables, teniendo en cuenta dos puntos:

- Hay que añadirle el modificador `final` antes del tipo de dato, sea el que sea.

- Por convención, el identificador de una constante, se escribirá entero en mayúsculas. Si tiene varias palabras, se usará guion bajo para separarlas.

```
1 final double PI = 3.141592653589793;
2 final String COLOR_ROJO = "\033[31m";
```

Entrada y salida de datos

Salida por pantalla

Para mostrar información por la consola, usaremos la función `System.out.println();`

```
1 String saludo = "Hola Mundo!";
2 System.out.println(saludo);
```

Dicha función imprimirá el resultado de la expresión, y un salto de línea al final.

Hay una versión que no imprimirá ningún salto de línea al final, que es `System.out.print();`.

También podemos incluir nosotros el salto de línea en la posición que queramos, incluyendo el carácter `\n` en cualquier parte del `String`.

```
1 System.out.print("\nHola\nMundo");
```

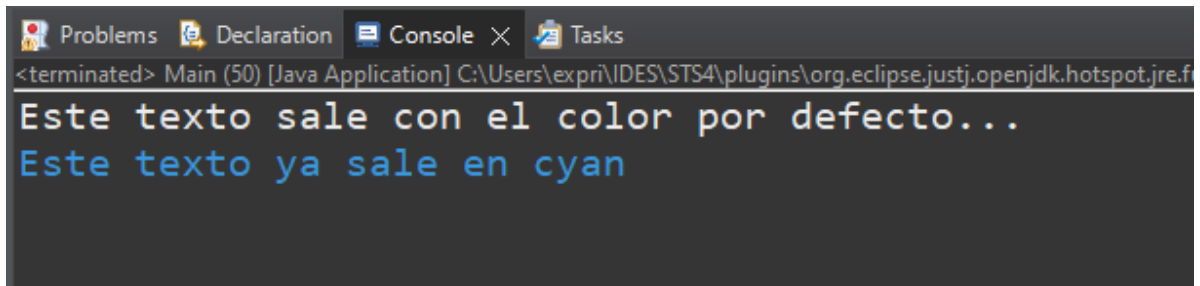
Esto imprimirá por consola:

```
1 Hola
2 Mundo
```

Colores

Es posible imprimir colores por la consola de Java en STS4. Tan sólo tendremos que imprimir un `String` concreto con un código de color y a partir de ese carácter imprimirá todo el texto en dicho color. A continuación la lista con los códigos de color y un ejemplo para imprimir un color concreto:

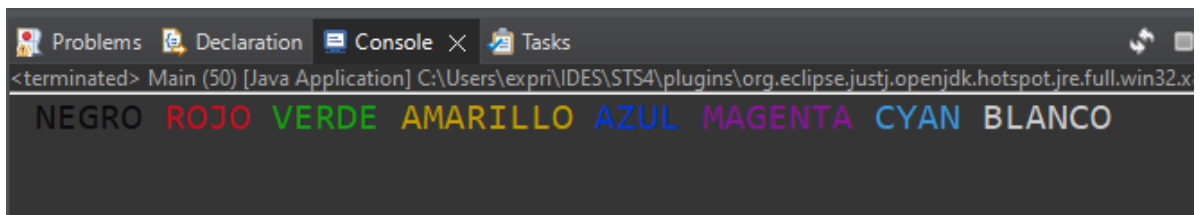
```
1 //Lista de colores disponibles
2 String black = "\033[30m";
3 String red = "\033[31m";
4 String green = "\033[32m";
5 String yellow = "\033[33m";
6 String blue = "\033[34m";
7 String purple = "\033[35m";
8 String cyan = "\033[36m";
9 String white = "\033[37m";
10 String reset = "\u001B[0m";
11
12 System.out.print(reset); //Ponemos el color por defecto
13 System.out.println("Este texto sale con el color por defecto...");
14 System.out.print(blue); //A partir de aquí imprimimos en azul
15 System.out.print("Este texto ya sale en azul");
```



Nota: Si cambiamos los colores, permanecerán activos hasta que se reseteen. Por lo que sería una buena práctica imprimir siempre al inicio del programa el color `\u001B[0m` (como hacemos en la línea 12 del ejemplo) para que no empezara a escribir con un color inesperado.

Aunque lo más fácil es guardar los códigos en variables para después imprimirlas cuando queramos, también podemos imprimir los códigos como parte de un String.

```
1 System.out.print("\033[30m NEGRO");
2 System.out.print("\033[31m ROJO");
3 System.out.print("\033[32m VERDE");
4 System.out.print("\033[33m AMARILLO");
5 System.out.print("\033[34m AZUL");
6 System.out.print("\033[35m MAGENTA");
7 System.out.print("\033[36m CYAN");
8 System.out.print("\033[37m BLANCO");
```



También es posible cambiar el color de fondo y algunos estilos básicos como negrita o cursiva, aunque dependerá de la compatibilidad de la versión de Eclipse respecto a los códigos ANSI.

La lista completa de códigos de escape para los colores ANSI la puedes encontrar en el siguiente enlace: https://en.wikipedia.org/wiki/ANSI_escape_code#Colors

Entrada por teclado

Para leer datos desde teclado en un programa java, usaremos la clase `Scanner`. Primero instanciaremos un nuevo objeto de la clase `Scanner`.

```
1 Scanner teclado = new Scanner(System.in); //Creamos un objeto Scanner
```

Nota: La clase `Scanner` está en el paquete `java.util`. Deberemos importar el paquete para poder usar la clase.

Una vez creado el objeto `Scanner`, usaremos sus métodos para pedir el tipo de dato que necesitemos por teclado:


```
1 int edad = teclado.nextInt();
2 float media = teclado.nextFloat();
3 double valor = teclado.nextDouble();
4 String nombre = teclado.nextLine();
```

Cada vez que llamemos a un método de los anteriores, la ejecución de la aplicación se detendrá y esperará a que introduzcamos un valor por la consola. Al pulsar intro se terminará la lectura por teclado, el valor recogido será devuelto por el método, y se asignará en la variable.

No es necesario crear nuevas instancias de la clase Scanner, podemos reutilizar la misma todas las veces que necesitemos. En nuestro ejemplo, es el objeto llamado `teclado`.

Cerrar teclado

Si no cerramos el teclado, se produce una fuga de recursos y los editores nos pueden avisar con un `warning`, por lo que podemos cerrarlo para evitar dicho aviso. Para cerrar el teclado usaremos el método `.close()` de la clase `Scanner`.

```
1 Scanner teclado = new Scanner(System.in);
2 ...
3 teclado.close(); // Lo cerramos cuando ya no vamos a usarlo
```

Nota: Si cerramos el objeto scanner y después lo intentamos usar, provocará una

Buffer de teclado

Al pedir datos por teclado, se van almacenando los caracteres pulsados en un *buffer* (memoria intermedia). Al usar los métodos `.nextInt()`, `.nextDouble()`, `.nextLine()`, etc., lo que hacen es extraer *tokens* (trozos) de ese buffer.

Cada método extraerá lo que está buscando y lo devolverá. Pero tenemos un efecto curioso. Veamos la siguiente secuencia.

1. Al usar el método `.nextLine()`, se abre el buffer e introducimos `Max Power` y pulsamos intro, por lo que en el buffer habrá `Max Power\n`. El método extraerá todos los caracteres que encuentre hasta el `/n` y los devolverá en formato `String`.
2. Hasta aquí todo normal y controlado. Pero pidamos ahora un número.
3. Al usar el método `.nextInt()` (o cualquiera para números), se abre el buffer e introducimos `1234` y pulsamos intro, en el buffer habrá `1234\n`. El método extrae el siguiente número que encuentre en el buffer y lo devuelve con el tipo pedido (`int`, `double` o `float`).
Dejando el resto del buffer intacto. Es decir, dejaría en su interior un `\n` residual.
4. Si en el buffer queda un `\n`, y volvemos a hacer un `.nextLine()`, tendremos un efecto inesperado. El método irá al buffer y como no está vacío, no se detiene a pedirnos nada, extrae el `\n` y sigue con la ejecución del programa.

Problema: Resumiendo, **después de hacer** un `.nextInt()`, `.nextDouble()` o `.nextFloat()`, **si hacemos un** `.nextLine()`, el primero lo ignorará (los sucesivos si funcionarán). Si primero pedimos el `.nextLine()` y después el `.nextInt/Float/Double()`, no existirá problemas, por lo explicado anteriormente.

Solución: Si pedimos un número, y después queremos pedir un String (es importante que sea en ese orden), necesitaremos limpiar el buffer tras pedir el número, haciendo:

```
1 teclado.nextLine();
```

No es necesario que guardemos el resultado en ninguna variable (podemos dejar que se vaya al cielo de los `String`). Ejecutando esa línea, cuando hagamos el `.nextLine()` que necesitamos, se detendrá como debe, ya que el buffer estará vacío.

Aquí un ejemplo de todo lo explicado, secuencialmente:

```
1 Scanner teclado = new Scanner(System.in);
2 System.out.println("Introduce tu edad y tu nombre, en ese orden:");
3 int edad = teclado.nextInt(); //Se detiene para introducir una edad
4 String nombre = teclado.nextLine(); //Se lo salta, porque en el buffer quedó
  un \n de antes
5 String apodo = teclado.nextLine(); //Se detiene, porque el buffer ya se
  vació en la línea anterior
```

Y como podemos solucionarlo:

```
1 Scanner teclado = new Scanner(System.in);
2 System.out.println("Introduce tu edad y tu nombre, en ese orden:");
3 int edad = teclado.nextInt(); //Se detiene para introducir una edad
4 teclado.nextLine(); //Limpiamos el buffer, ya que había un \n
5 String nombre = teclado.nextLine(); //Se detiene, porque el buffer ya está
  vacío
6 String apodo = teclado.nextLine(); //Se detiene de nuevo, porque el buffer
  está vacío
```

🤖: También podemos solucionarlo pidiendo los `String` en primer lugar, y después los números. O bien usando una nueva instancia de la clase `Scanner`, la cual tenga un buffer vacío.