PROGRAMACIÓN UD-2

SINTAXIS Y CONDICIONALES

ÍNDICE

[INTRODUCCIÓN 3](#_Toc117447962)

[A. HISTORIA DE JAVA 3](#_Toc117447963)

[B. CARACTERÍSTICAS 3](#_Toc117447964)

[C. COMPILACIÓN Y EJECUCIÓN 4](#_Toc117447965)

[EDICIONES DE JAVA 4](#_Toc117447966)

[APPLICATION PROGRAMING INTERFACE (API) JAVA 5](#_Toc117447967)

[PROGRAMA PRINCIPAL 5](#_Toc117447968)

[SINTAXIS JAVA 6](#_Toc117447969)

[A. VARIABLES 6](#_Toc117447970)

[1. TIPOS PRIMITIVOS JAVA 7](#_Toc117447971)

[B. COMENTARIOS 8](#_Toc117447972)

[C. ENTRADA Y SALIDA POR CONSOLA 8](#_Toc117447973)

[D. OPERADORES 10](#_Toc117447974)

[1. OPERADORES DE ASIGNACIÓN 10](#_Toc117447975)

[2. OPERADOR TERNARIO 11](#_Toc117447976)

[3. OPERADORES ARITMÉTICOS 11](#_Toc117447977)

[4. OPERADORES RELACIONALES 12](#_Toc117447978)

[5. OPERADORES LÓGICOS 12](#_Toc117447979)

[6. OPERADORES ORIENTADOS A BIT 13](#_Toc117447980)

[E. CONVERSIÓN DE TIPOS 15](#_Toc117447981)

[F. CONDICIONALES 17](#_Toc117447982)

[1. CONDICIONAL SIMPLE: IF 17](#_Toc117447983)

[2. CONDICIONAL DOBLE: IF ELSE 17](#_Toc117447984)

[3. OPERADOR TERNARIO 18](#_Toc117447985)

[4. ANIDAMIENTO DE CONDICIONALES 18](#_Toc117447986)

[5. CONDICIONAL MÚLTIPLE: SWITCH 18](#_Toc117447987)

# INTRODUCCIÓN

## HISTORIA DE JAVA

Sun Microsystems crea en 1991 el lenguaje Oak (recibe el nombre porque Gosling tenía un roble fuera de su oficina) pero el nombre ya estaba registrado y se utilizó Green en su lugar. Al ser un nombre poco comercial se terminó decidiendo que se llamaría JAVA. 2 teorías:

* El nombre es por los creadores: James Gosling (J), Arthur Van Hoff (A y V) y Andy Bechtolsheim (A).
* El nombre (y logo) se debe al nombre que recibe al café en USA (java).

Java se crea con la finalidad de crear una televisión interactiva, pero esto sólo se llegó a utilizar internamente en la empresa. Finalmente se decidió que la finalidad sería crear un lenguaje que fuera independiente de la plataforma (lenguaje multiplataforma) para uso en diferentes dispositivos electrónicos.

Es decir, lo que se buscaba era solucionar el principal problema de C++: el ser un lenguaje que sólo compila y ejecuta en la plataforma en la que se desarrollaba. Debido a la naturaleza cambiante de los pequeños dispositivos electrónicos, Sun quería crear un lenguaje que funcionará en todos ellos independientemente de los cambios.

## CARACTERÍSTICAS

Aunque es muy similar a C y C++ tiene varias diferencias:

* **No existen punteros**. Los punteros son variables donde se almacena una dirección de memoria (por ejemplo, el dato “23” estaría asignado a la posición de memoria 01001), al eliminarlos se aumenta la seguridad y la facilidad de uso.
* Es un lenguaje completamente **orientado a los objetos**.
* Está preparado para las **redes TCP/IP** y especialmente Internet.
* Implementa **excepciones** de forma nativa. Las excepciones son la forma que tiene el lenguaje de controlar la aparición de errores.
* Es un **lenguaje interpretado**. El ser un lenguaje interpretado significa que, al iniciarlo, ejecuta directamente las instrucciones sin una previa compilación a lenguaje máquina (unos y ceros). Esto acelera su ejecución remota, pero provoca que las aplicaciones de Java sean más lentas.
* Permite **múltiples hilos** (tareas) **de ejecución** (paralela).
* Admite **firmas digitales**.
* Es más **riguroso** que C y C++ en los tipos de datos y en su **sintaxis**. Esto hace que sea más sencillo el gestionar la aparición de errores.
* Es un **lenguaje multiplataforma**. Es decir, se puede ejecutar en cualquier sistema (Windows, Linux, OSX…) que cuente con la máquina virtual de Java.

## COMPILACIÓN Y EJECUCIÓN

En Java el código no se traduce a lenguaje máquina, sino que se produce un proceso llamado **precompilación** y que produce un archivo (de extensión .class) que contiene código que no es directamente ejecutable (**código bytecode o J-Code**).

El **archivo CLASS** no es ejecutable por el sistema con un simple doble click, sino que debe ser interpretado por una aplicación, la ***Java Virtual Machine* (JVM)**. Actualmente se conoce como***Java Runtime Environment* (JRE)**.

La ventaja de este proceso es que convierte a Java en un lenguaje multiplataforma ya que ese archivo CLASS se podrá ejecutar en cualquier sistema que cuente con la JRE. La desventaja es que, si el código se programa en X versión de Java, el JRE del ordenador donde se ejecute debe ser de al menos esa versión (por ejemplo, si se programa en Java ver.1.11, el JRE donde se ejecute debe ser al menos la ver. 1.11).

Esta forma de producir el código Java recibe el nombre de **JIT (*Just In Time*)** ya que el código ejecutable se produce justo en el instante de ejecución del programa (no existe en ningún momento código ejecutable).

# EDICIONES DE JAVA

Actualmente tenemos tres ediciones de Java (en función de la versión la API será más o menos extensa):

* **Java SE (*Standard Edition*)**: Incorpora los elementos necesarios para crear **aplicaciones de escritorio** con o sin interfaz gráfica de usuario (IGU o GUI), acceso al sistema de archivos, comunicación a través de redes, concurrencia y otros servicios básicos.
* **Java EE (*Enterprise Edition*)**: Se utiliza para el desarrollo de software que se ejecutará en un **servidor de aplicaciones**. Además de lo que incorpora SE agrega servicios para gestionar la persistencia de objetos en BBDD, hacer posible la invocación remota de métodos, crear aplicaciones con interfaz de usuario web, etc.
* **Java ME (*Micro Edition*)**: Se utiliza para el desarrollo de aplicaciones para **sistemas con recursos limitados** como los smartphones, los electrodomésticos… o para el uso en equipos con entornos embebidos (sistemas informáticos basados en un procesador que ejecutará pocas funciones dedicadas) como por ejemplo una Rasperry Pi.

Antiguamente había una cuarta versión, la **Java FX** que era una versión con aceleración gráfica, pero desde Java 7 se incluye en la versión SE.

# APPLICATION PROGRAMING INTERFACE (API) JAVA

Es la interfaz de programación de aplicaciones de Java creada por los autores del lenguaje Java y que da a los programadores medios para desarrollar aplicaciones Java.

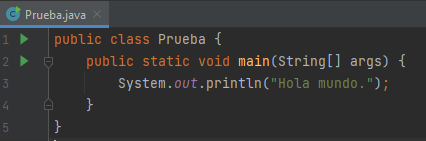
Debido a la naturaleza de Java como lenguaje orientado a objetos, la API tiene una lista de clases utilitarias para realizar todas las tareas necesarias dentro de un programa.

Se organiza en paquetes que contienen un conjunto de clases relacionadas semánticamente. Esto provoca que dentro de dos paquetes diferentes podamos tener dos clases con el mismo nombre pero que realicen funciones distintas.

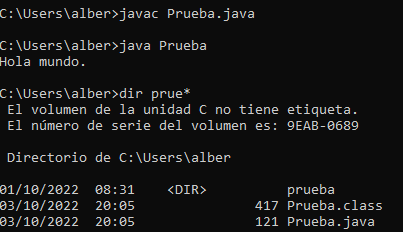
No sólo existe la API propia de Oracle (API JDK) sino que existen multitud de API propietarias de otras empresas. En los IDEs de Java se incluye por defecto la JDK, pero si se utilizan otras APIs habrá que referenciales en el código.

# PROGRAMA PRINCIPAL

Siempre que se cree un programa ejecutable en Java se tendrá una estructura como la siguiente:



En la pestaña se puede ver el **nombre del archivo** (Prueba.java) que es el mismo que el de la **clase** (public class prueba). El nombre de la clase siempre va con mayúscula.

En la línea 2 de código se encuentra el **método principal (o main)** que será la clase ejecutable del programa y dentro de este (línea 3) una **sentencia de tipo invocación a método** (en este caso imprimir por consola el mensaje “Hola mundo”).

Para sacar el bytecode de Java hay que ejecutar javac nombre\_clase.java. Para ejecutar el programa, pondremos java nombre\_clase.

# SINTAXIS JAVA

## VARIABLES

Es un identificador que contiene un valor que puede variar a lo largo de la ejecución del programa.

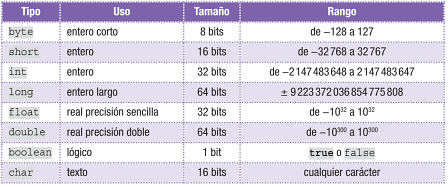
Al ser Java un lenguaje key sensitive hay que tener cuidado con las mayúsculas y minúsculas (“nota” y “Nota” serán interpretados por Java como dos variables distintas). Como normas generales tenemos:

* Deben comenzar por una letra, una barra baja (\_) o el símbolo del dólar ($).
* El resto de los caracteres pueden ser números, letras, \_ o $.

Como recomendaciones sintácticas tenemos:

* Si la variable se compone de varias palabras es recomendable escribir cada palabra (menos la primera) con mayúscula. Por ejemplo, “notaAlumno” (sería una variable de tipo double) o “aprobado” (sería una variable de tipo boolean).
* Si una variable es de tipo básico, es recomendable colocar delante un identificador de dicho tipo. Por ejemplo, “rlnNotaAlumno” o “blnAprobado”.

Se deben optimizar las variables, es decir, elegir aquellas que ocupen el espacio óptimo en memoria para almacenar lo que se quiere guardar.



Abreviaciones:

* byte = byt.
* short, int y long = int.
* float y double = rln.
* boolean = bln.
* char = chr.

Ejemplos de declaración de variables:



Lo primero que se escribe es el tipo de variable, luego el nombre de la variable (precedida por un identificador del tipo) y, por último, el valor de inicio que tomará esta variable. Este valor inicial puede escribirse o no, si no se escribe la variable tomará el valor por defecto.

Si al inicio de la declaración de variable (antes del tipo) se coloca el modificador final, se convertirá la variable en una constante y no podrá cambiar su valor a lo largo de la ejecución del programa. Se recomienda escribir las constantes con mayúsculas.

1. TIPOS PRIMITIVOS JAVA

Si se excede el valor máximo que puede tomar una variable, esta tomará el siguiente de manera circular. Por ejemplo:



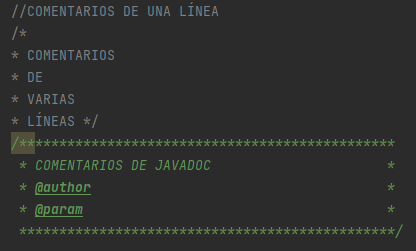
La variable “bytValor” pasará a tener un valor de -128 ya que su valor máximo es 127.

## COMENTARIOS

Para facilitar el entendimiento del código por parte de otros usuarios es importante escribir comentarios (siempre teniendo en cuenta que el exceso de comentarios es tan problemático como su carencia).

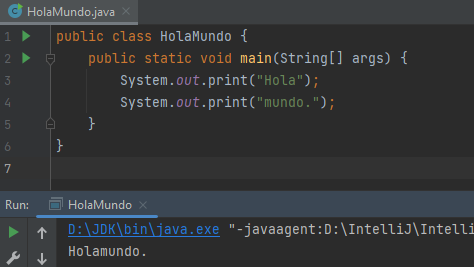
Existen dos tipos de comentarios.

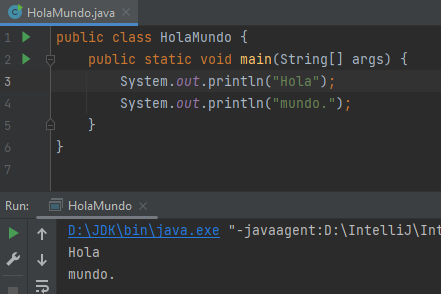
* **Comentarios de una línea**. Se preceden de una doble barra inclinada (//) y no tienen cierre. Se usan por ejemplo para comentar el uso de una variable.
* **Comentarios multilínea**. Se preceden de una barra inclinada de un asterisco y se cierran con un asterisco y una barra inclinada (/\* \*/). Se usan para explicar un procedimiento o el uso de una clase.

A su vez se pueden insertar **comentarios para generar documentación automática** (conocida como JavaDoc). Para ello se utilizarán comentarios de múltiples líneas, pero con una estructura diferente:

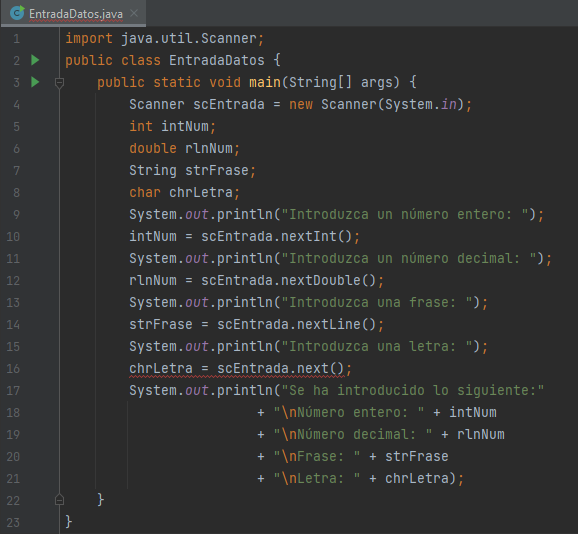
* Comenzarán con una barra y dos asteriscos (/\*\*).
* Utilizarán un identificativo del tipo de comentario precedido de una arroba (@authot, @version, @param…).

## ENTRADA Y SALIDA POR CONSOLA

* **Salida**: Para la salida de datos por consola tenemos dos métodos principales dentro de la clase System.
  + **System.out.print(“Mensaje”)**. Imprime sin salto de línea.
  + **System.out.println(“Mensaje”)**. Imprime con salto de línea.



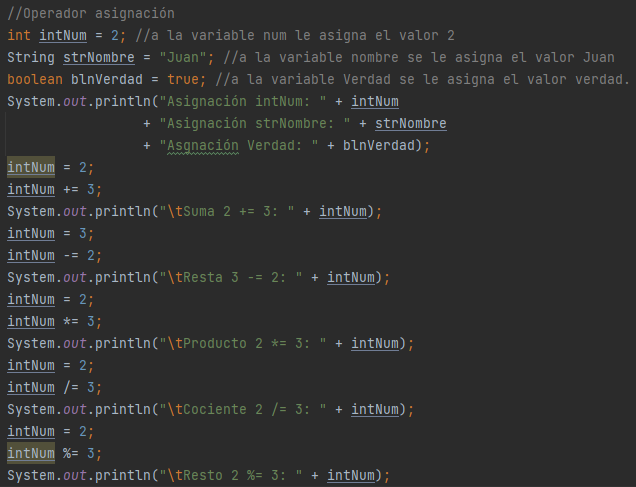
* **Entrada**: Para la entrada de datos por teclado existen varias opciones, una de ellas es la clase Scanner que, a su vez, tiene varios métodos:
  + **.nextInt()**: lee un número entero.
  + **.nextDouble():** lee un número real.
  + **.nextLine()**: lee todos los caracteres hasta que se pulsa intro.
  + **.next()**: lee todos los caracteres hasta que se pulsa intro, tabulado o un espacio en blanco.



## OPERADORES

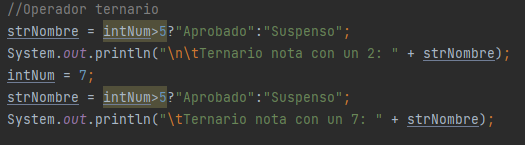
1. OPERADORES DE ASIGNACIÓN

|  |  |  |
| --- | --- | --- |
| TIPO | OPERADOR | SIGNIFICADO |
| Asignación | = | Modifica el valor de una variable. |
| += | Suma el valor de la izquierda con el que se encuentra a la derecha (2+=3 sería lo mismo que 2+3). |
| -= | Resta el valor de la izquierda con el que se encuentra a la derecha (3-=2 sería lo mismo que 3-2). |
| \*= | Multiplica el valor de la izquierda con el que se encuentra a la derecha (2\*=3 sería lo mismo que 2\*3). |
| /= | Divide el valor de la izquierda con el que se encuentra a la derecha (2/=3 sería lo mismo que 2/3). |
| %= | Calcula el resto de la división entera entre el valor de la izquierda y el de la derecha (2%=3 sería lo mismo que 2%3). |



1. OPERADOR TERNARIO

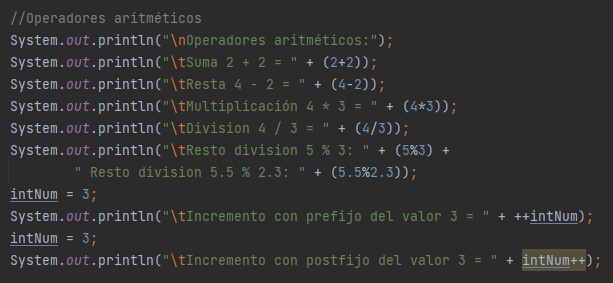
|  |  |  |
| --- | --- | --- |
| TIPO | OPERADOR | SIGNIFICADO |
| Ternario | =? | Asigna un valor entre dos posibles en un condicional. |



1. OPERADORES ARITMÉTICOS

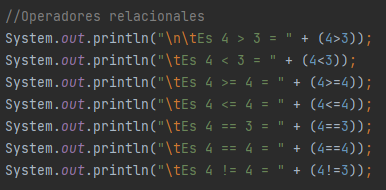
|  |  |  |
| --- | --- | --- |
| TIPO | OPERADOR | SIGNIFICADO |
| Aritméticos  (actúan sobre números) | + | Suma operandos.  Concatena cadenas.  Indica que un valor es positivo. |
| - | Resta operandos.  Indica que un valor es negativo.  Cambio de signo algebraico. |
| \* | Multiplica operandos. |
| / | Divide operandos. |
| % | Resto de la división entera. |
| ++ | Suma 1 al operando.\* |
| -- | Resta 1 al operando.\* |

\*Se pueden usar como prefijo (++contador) o sufijo (contador++). En el primer caso, la variable se aumenta antes de que sea utilizada. En el segundo caso la variable se aumenta después de ser utilizada.



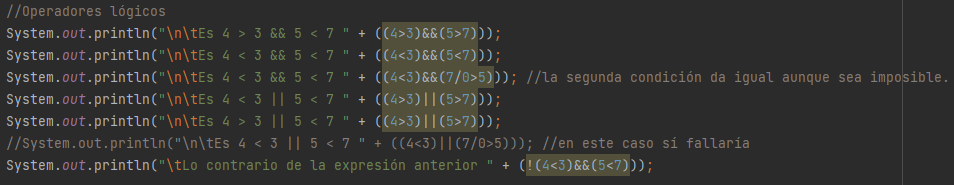
1. OPERADORES RELACIONALES

|  |  |  |
| --- | --- | --- |
| TIPO | OPERADOR | SIGNIFICADO |
| Relacionales  (devuelven un booleano) | > | Mayor que. |
| >= | Mayor o igual que. |
| < | Menor que. |
| <= | Menor o igual que. |
| == | Igual que. |
| != | Distinto que. |



1. OPERADORES LÓGICOS

|  |  |  |
| --- | --- | --- |
| TIPO | OPERADOR | SIGNIFICADO |
| Lógicos  (permiten generar expresiones complejas) | && | Ambas expresiones son verdaderas. |
| || | Una de las expresiones es verdadera. |
| ! | La expresión de la derecha es falsa. |



1. OPERADORES ORIENTADOS A BIT

Para pasar de base 10 a base 2 (binario) un número hay que dividir el número entre 2 hasta que el divisor sea menor que el dividendo y, entonces, juntar los restos empezando desde abajo, por ejemplo, para pasar el 14 a binario habría que hacer lo siguiente:

14 / 2 = 7 (resto 0)

7 / 2 = 3 (resto 1)

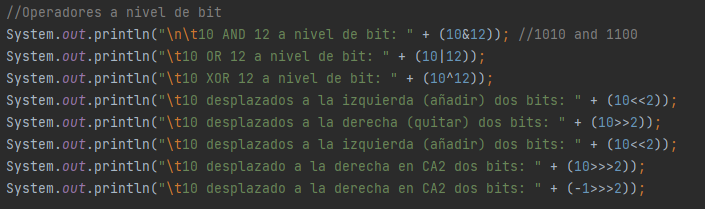
3 / 2 = 1 (resto 1)

1 / 2 = 0 (resto 1)

El número en binario sería por tanto el 1110.

Además, tenemos que tener en cuenta que previo al número binario habrá un dígito más en función de si es positivo (llevará un 0) o negativo (llevará un 1).

|  |  |  |
| --- | --- | --- |
| TIPO | OPERADOR | SIGNIFICADO |
| Orientados a bit | & | AND a nivel bit. |
| | | OR a nivel bit. |
| ^ | XOR a nivel bit |
| << | Desplazamiento a la izquierda a nivel de bit. |
| >> | Desplazamiento a la derecha a nivel de bit. |
| <<< | Desplazamiento a la izquierda a nivel de bit en CA2. |
| >>> | Desplazamiento a la derecha a nivel de bit en CA2. |



En el operador AND es necesario que ambas opciones sean verdaderas para que sea verdadero (es decir, un 1) en cualquier otro caso será falso (es decir, un 0). Por ejemplo,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NUMERO  DECIMAL | NÚMERO  BINARIO | | | |
| 10 | 1 | 0 | 1 | 0 |
| 12 | 1 | 1 | 0 | 0 |
|  | 1 | 0 | 0 | 0 |

Como sólo la primera pareja de dígitos es Verdadero-Verdadero y el resto son Falso-Verdadero, Verdadero-Falso y Falso-Falso, el número binario resultante es 1000, que pasado a base 10 es el 8. Para calcularlo hay que seguir el siguiente cálculo:

(1 \* 23 + 0 \* 22 + 0 \* 21 + 0 \* 20) = 8 + 0 + 0 + 0 = 8

En el operador OR sólo es necesario que una de las opciones sea verdadera para ser verdadero. Usando los mismos números:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NUMERO  DECIMAL | NÚMERO  BINARIO | | | |
| 10 | 1 | 0 | 1 | 0 |
| 12 | 1 | 1 | 0 | 0 |
|  | 1 | 1 | 1 | 0 |

En este caso el número resultante es el 1110 (Verdadero-Verdadero, Falso-Verdadero, Verdadero-Falso y Falso-Falso), que pasado a base 10 es el 14. Mismo cálculo que antes:

(1 \* 23 + 1 \* 22 + 1 \* 21 + 0 \* 20) = 8 + 4 + 2 + 0 = 14

En el operador XOR (u OR exclusivo) dará como salida un verdadero (1) siempre que sólo una de las entradas sea 1. Usando los mismos números:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NUMERO  DECIMAL | NÚMERO  BINARIO | | | |
| 10 | 1 | 0 | 1 | 0 |
| 12 | 1 | 1 | 0 | 0 |
|  | 0 | 1 | 1 | 0 |

En este caso el número resultante es el 0110 ya que sólo los dos números de en medio tienen una única entrada verdadera. Pasado a base 10 es el 6. Mismo cálculo:

(0 \* 23 + 1 \* 22 + 1 \* 21 + 0 \* 20) = 0 + 4 + 2 + 0 = 6

En el operador de desplazamiento a la izquierda a nivel de bit (<<) lo que se hace es añadir X bits al número inicial. Por ejemplo, en la expresión 10<<2, estamos diciendo que al número 10 en binario (1010) hay que añadirle 2 bits (00), dando como resultado el 101000, es decir, el 40 en base 10.

(1 \* 25 + 0 \* 24 + 1 \* 23 + 0 \* 22 + 0 \* 21 + 0 \* 20) =

= 32 + 0 + 8 + 0 + 0 = 40

En el operador de desplazamiento a la derecha a nivel de bit (>>) se hace lo contrario, se quitan X bits al número inicial. Por ejemplo, en la expresión 10>>2, vamos a quitarle 2 bits al número 10, es decir, va a pasar de 1010 a 10 (2 en base 10).

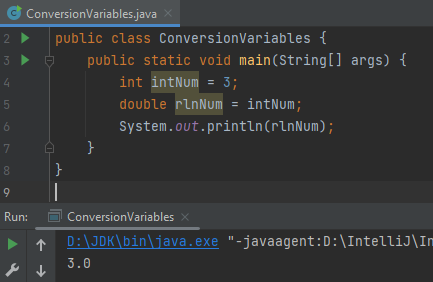
(1 \* 21 + 0 \* 20) = 2

## CONVERSIÓN DE TIPOS

En Java existe la posibilidad de cambiar el tipo de una variable en otro diferente al que originalmente fue declarado. Esto se denomina conversión o tipado.

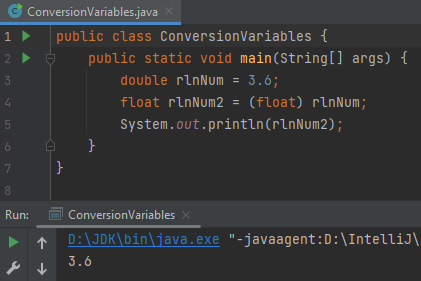
Existen dos tipos de conversiones:

* **Conversión implícita.** La realiza el propio compilador cuando se encuentra una expresión con variables de diferente tipo. Siempre convierte las variables al mayor de los tipos (el que más espacio ocupe en memoria).



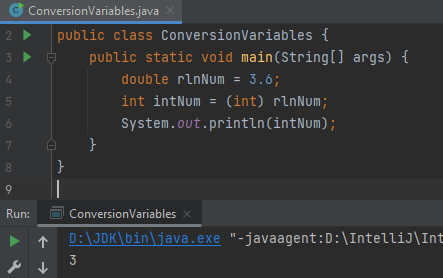
En este caso Java automáticamente cambiar el valor del int 3 en double 3.0 antes de asignarlo a la variable rlnNum e imprimirlo por consola. Java es capaz de hacer esto automáticamente porque la variable int ocupa originalmente 32 bytes de memoria por lo que entra sin problemas en un double de 64 bytes.

* **Conversión explícita**. La realiza el programador y consiste en colocar el nuevo tipo entre paréntesis, a la izquierda del valor que queremos convertir.



En este caso Java no convertirá automáticamente el double 3.6 en un float 3.6 porque el tipo double ocupa el doble que el int (64 bytes vs 32 bytes) y necesitará de un casteo o tipeo por parte del programador para cambiar la variable. Una vez se haya efectuado el tipeo se imprimirá por consola un 3.6.

Es importante tener en cuenta el tamaño de los tipos en los que se quiere convertir una variable. No todos los tipos son compatibles entre sí y se produce pérdida de información. Esto se denomina conversión no segura.



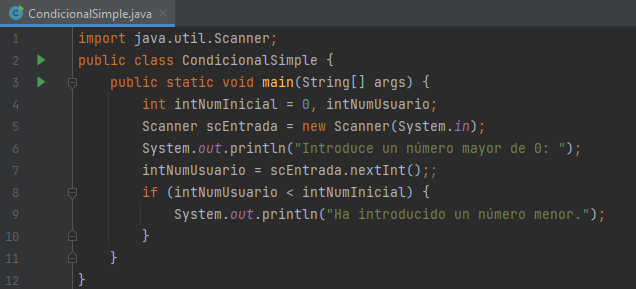
Para realizar conversiones seguras es importante tener en cuenta la siguiente tabla de conversiones para que no se pierda información:

|  |  |
| --- | --- |
| TIPO ORIGEN | TIPO CASTEO |
| byte | double, float, long, int, char, short |
| short | double, float, long, int |
| char | double, float, long, int |
| int | double, float, long |
| long | double, float |
| float | double |

## CONDICIONALES

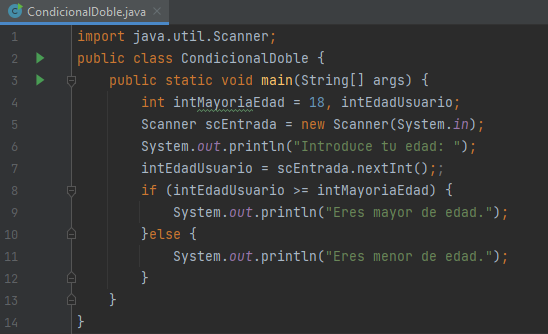
1. CONDICIONAL SIMPLE: IF

La sentencia if proporciona control sobre un conjunto de instrucciones que pueden ejecutarse o no dependiendo de la evaluación de una condición.



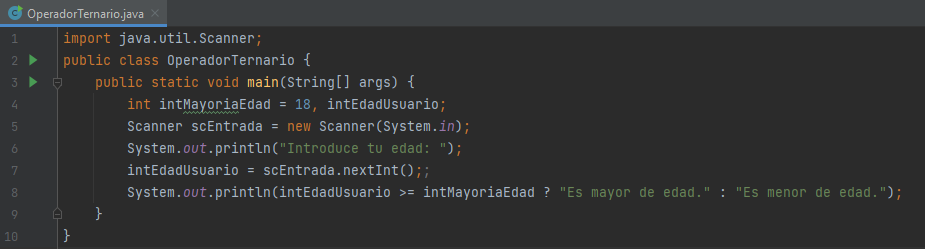
Los bloques de instrucciones van encerrados entre llaves (si es sólo una instrucción no es necesario). Las variables que se creen dentro de un bloque quedan “encerradas” en el bloque.

1. CONDICIONAL DOBLE: IF ELSE

Diferencia un grupo de acciones para la condición verdadera y un conjunto de acciones para la condición falsa.

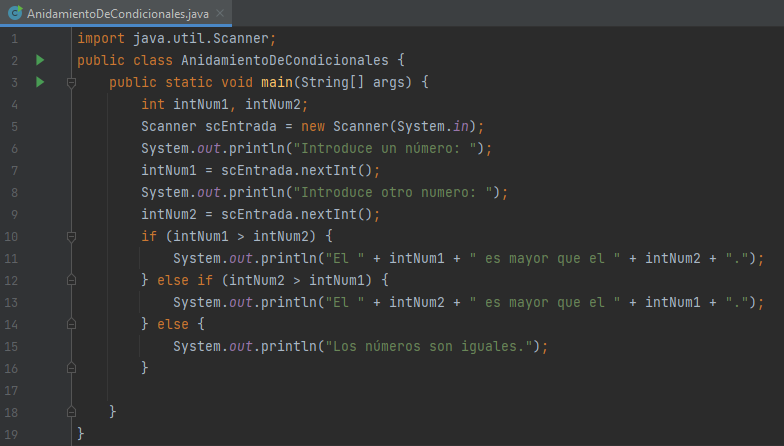
1. OPERADOR TERNARIO

Sustituye un condicional doble si en sus bloques se va a realizar una asignación única a una misma variable.

Es recomendable usarlo porque ocupa menos espacio y es más legible.

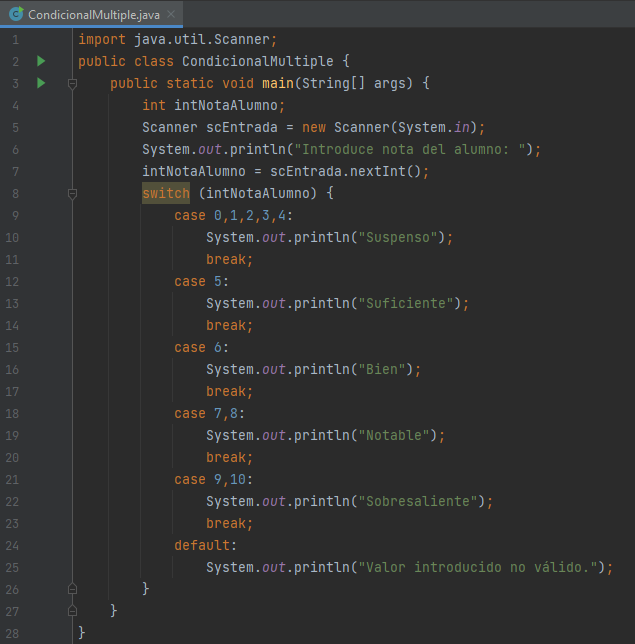
1. ANIDAMIENTO DE CONDICIONALES

En caso de necesitar realizar múltiples comprobaciones se pueden usar los operadores lógicos, pero también es posible anidar múltiples sentencias if o if-else.



1. CONDICIONAL MÚLTIPLE: SWITCH

Se utiliza cuando existen varios if o if-else que, anidados, harían el código poco legible. En estos casos se utiliza el condicional de opción múltiple o switch.

Consiste en una serie de opciones (cases) que ejecutan unas instrucciones u otras. Cada case se debe cerrar con un break que corte la ejecución del switch (si no existiera el break se ejecutaría todo el código posterior al case que cumple la condición). Al final de todos los cases se coloca una instrucción default que realiza una serie de instrucciones si ninguna de las opciones se puede ejecutar.