# Curso básico de Java SE 2018

En este curso se revisan los fundamentos de la programación en Java Standard Edition.

## Ventajas de aprender Java:

* **Empezar rápidamente:**

- Esto es posible gracias a que es un lenguaje de programación similar a C y C++

* **Escribir menos código:**

- Gracias a que es enteramente OOP y permite la reutilización de código.

* **Escribir mejor código:**

- Con el curso de Java básico aprendemos buenas prácticas de codificación.

## ¿Qué es Java?

Es uno de los lenguajes más utilizados a nivel mundial.

También es un lenguaje de programación de alto nivel.

El lema de java es: ***“Write Once, Run Anywhere” o WORA***

Esto significa que un código de Java puede ser leído en cualquier SO, computadora, etc.

## Java también cumple con ciertas características que lo hacen un excelente lenguaje.

Estas características son:

* **Es simple:**

- Basado en C, sintaxis muy parecida a C y C++ (OOP).

- Se hereda de una sola clase.

- Cuenta con Garbage Colector, el cual se encarga de remover los objetos que no están en uso para liberarlos de la memoria y así hacer más eficiente el lenguaje.

* **Orientada a Objetos:**

- APRENDER A FONDO!

- Java como tal es un lenguaje de programación orientada a objetos (OOP).

* **Distribuido:**

- Está diseñado para trabajar con protocolos TCP/IP, HTTP, FTP, etc. Todo lo necesario para trabajar en ambientes de redes.

* **Arquitectura neutral:**

- Corre diversos ambientes de trabajo (no solo Windows, y Linux).

* **Es portable.**

- o sea, el mismo código corre en varios sistemas operativos.

* **Alto desempeño:**

- Es compilado e interpretado.

* **Seguro:**

- Gracias a la máquina virtual de Java (JVM).

- El código no está expuesto a nadie, ya que a la hora de compilarlo, JVM lo convierte a ByteCode (archivo .class) y a la hora de correr el programa no lee el código fuente.

## Componentes de Java:

***JDK***: Java Development Kit.

***JRE***: Java Runtime Environment. Este componente nos permite ejecutar los programas como tal.

## Java Virtual Machine

**¿Cómo funciona la máquina virtual de Java?**

Al ser un lenguaje de alto nivel, Java usa un código intermediario para poder comunicarse con el hardware, este es el **ByteCode.**

La máquina virtual de Java lee el ByteCode y lo traduce al lenguaje máquina del sistema operativo donde se está trabajando.

# Tipos de datos

## Tipos de datos primitivos enteros

**Variable**: una variable es un espacio en memoria al que le asignamos un contenido; puede ser un valor numéricp, de tipo carácter o cadena de caracteres.

Tipos de datos en Java:

Tipo primitivo:

* **Enteros:**

- Son los que comienzan con letra minúscula

* + bite: 1 byte
  + short: 2 bytes
  + int: 4 bytes
  + long: 8 bytes
* **Punto flotante (con punto decimal):**
  + float: 4 bytes
  + double: 8 bytes
* **Tipo de dato de texto (sólo una letra):**
  + char: rango Unicode; 2 bytes
* **Tipos de datos lógicos:**
  + boolean : true o false; 2 bytes

## Naming en Java

Java es sensible a mayúsculas y minúsculas, tanto como a guion bajo y $,

**Comenzar con letra, $ ó \_:** Al incluir el símbolo antes del nombre de nuestra variable, la hará diferente, aunque existan otras muy similares; por ejemplo: variable, Variable, \_variable y $variable son únicas, cada una de ellas es diferente a las demás.

**Letras posteriores:** Podemos utilizar estos mismos símbolos y números al final del nombre de la variable, sin embargo, no debemos escribir números al inicio de la variable; por ejemplo: int variable1 = 1, int 1variable =1, la primera variable es correcta la segunda mostraría un error de sintaxis.

**Las constantes:** se deben escribir con mayúsculas y contienen “\_”

Por convención se debe utilizar la técnica “camello” o camel case, esta tendrá dos formas:

* **Upper Camel case:** Serán el nombre nuestra clase y esto significa que siempre van a contener mayúsculas la primera letra, también los nombres de los archivos (siempre debe ser el mismo el nombre de la clase y del archivo)
* **Lower Camel case:** Se utilizará para nombres de variables, de objetos y de métodos, estos siempre comenzarán en mayúscula

## Cast

En la programación existen situaciones en las que es necesario cambiar el tipo de dato, cuando necesitamos que un tipo se comporte como otro, en Java esto se conoce como Cast y genera un tipo de dato diferente al original

Podemos ejecutar casteos de:

* Tipos de datos primitivos
* Tipos de datos objeto primitivos

**Sintaxis:**

double d = 86.45;  
 ✓int i = (int) d;

Es facil reconocer un cast, cuando vemos encerrado entre paréntesis un tipo de dato, significa que ahora estamos definiendo la variable “d” como un valor entero, la cual habíamos definido originalmente como un valor flotante.

A la hora de hacer casteos entre tipos de datos, es necesario tomar en cuenta el consumo de memoria de cada uno de ellos, esto quiere decir que, si requerimos castear o “transformar” una variable de tipo byte a short, Java lo hace sin problema alguno (sintaxis a continuación), mas, si debemos castear un dato de tipo short a uno de tipo byte debemos hacerlo con la sintaxis antes mencionada.

**Sintaxis:**

byte b = 4;

✓short s = b;

🗶b = S;🡨 en este caso, saldría un mensaje de “necesitas añadir un casteo”

Así es como Java realiza ese cast sin ningún problema.

Nota: al momento de castear una variable de tipo flotante i.e. doble a una de tipo entero, Java automáticamente “ignora” los números hallados después del punto decimal, dado a que este tipo de dato no reconoce los datos de tipo flotante.

También existen conversiones **NO seguras**. Esto se debe a que no todos los cast de variables se pueden realizar de forma segura, por ejemplo:

Al convertir un short de 2 bytes a byte, que contiene 1 solo byte, se deberá cortar los bits superiores, ocasionando pérdidas de información.

short datoshort = 259;

byte datobyte = (byte) datoshort;

Al imprimir la variable “datobyte” da como resultado 3.

Esto es porque 259, en binario es 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1, que equivale a 2 bytes.

Al castearlo en byte, toma los primeros 8 bits del lado derecho: 0 0 0 0 0 0 1 1. Este binario en forma decimal corresponde a 3.

## Arrays

Los arreglos se pueden definir como **objetos** en los que podemos guardar **más de una variable**.

Esto funciona como una “caja de galletas” en donde podemos almacenar más de una galleta, entonces, cada galleta es una variable y la caja en sí sería el arreglo.

Se pueden tener arreglos de n dimensiones. Esto no es, como tal, debido a la cantidad de elementos dentro del arreglo, sino a la forma en la que estén acomodados, por ejemplo:

Si los elementos están acomodados en una sola fila, obtenemos un arreglo de una dimensión, sin importar cuántos elementos habiten en esa fila.

Entonces, se puede entender a las dimensiones de un arreglo como los ejes de un plano cartesiano; las filas de nuestro arreglo serían las ordenadas de nuestro plano cartesiano y los elementos contenidos en esas filas, las abscisas o viceversa. Continuando con esta lógica, podemos entender una tercera dimensión de arreglos como los valores de un eje Z, el cual contiene varios de estos “planos” bidimensionales stackeados uno sobre el otro, como si de muñequitas rusas se tratara. Ahora bien, si hablamos de n dimensiones, no tenemos un referente claro en la vida real para demostrar más de tres dimensiones.

Así que, las dimensiones en los arreglos de más de tres significan que **dentro** de cada arreglo, de cada fila en cada dimensión debemos tener la misma cantidad de arreglos nuevos, justo como se muestra en la siguiente ilustración:

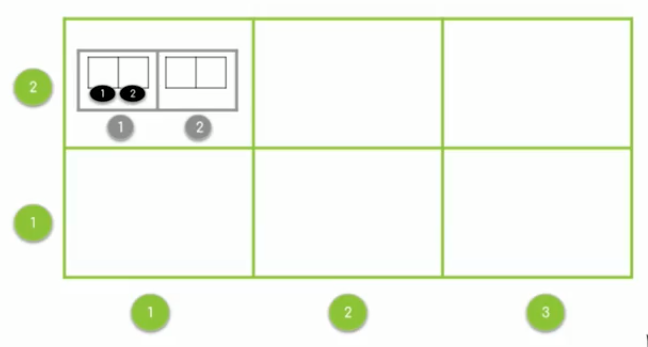


Ilustración : los arreglos de color gris (2D) deben estar también dentro de los demás arreglos en color verde (3D), hospedando la misma cantidad de arreglos en negro. De este modo, estamos hablando de un arreglo de n dimensiones.

### Declarar Arrays en Java

Existen dos formas de declarar los arrays en Java.

TipoDato[] nombreVariable; 🡨 cuando colocamos los corchetes, el compilador entiende que ahora “nombreVariable” no es un simple dato primitivo, sino un arreglo.

TipoDato nombreVariable[]; 🡨 también es posible declarar un arreglo

colocando los corchetes después del nombre.

### Definir tamaño del Array

Es importante especificar siempre la capacidad de los arreglos, el tamaño. Si no lo definimos nunca podremos hacer uso de dicho arreglo.

nombreVariable = new TipoDato[capacidad];

TipoDato[][][] nombreVariable = new TipoDato[No.renglones][No. columnas][Altura];

En la primera parte (antes del “=”) definimos la cantidad de dimensiones que contendrá el arreglo y en la segunda le damos el valor de los renglones, las columnas y la altura, respectivamente que contendrá cada dimensión.

## Búsqueda de elementos dentro de los Arrays

Existe una forma de encontrar dentro de nuestros arrays los elementos que tenemos en cada una de las dimensiones.

Esto se logra haciendo uso de los ***índices***, estos buscan dentro de cada arreglo, una posición específica, según la definamos, por ejemplo:

Tenemos un elemento llamado “monkey” dentro de un arreglo de 4 dimensiones, al ser un arreglo de más de tres dimensiones, se puede complicar un poco, pues es necesario respetar la sintaxis mencionada anteriormente. En este caso, el código para encontrar a monkey dentro de nuestro arreglo sería:

char [][][][] monkey = new char[2][3][2][2];

monkey[1][0][0][1] = 'M';

System.out.println(monkey[1][0][0][1]);

Así pues, tenemos un arreglo de 4 dimensiones, las cuales se componen por 2 renglones, 3 columnas y dos niveles que contienen 2 elementos dentro. Entonces, podemos ver que “monkey” es el primer elemento dentro del primer nivel que habita en el primer renglón de la primera columna [1][0][0][1].

Nota: Los índices comienzan desde 0 hasta n-1, esto quiere decir que nuestro primer renglón o columna o dimensión estará ubicada dentro del arreglo en “0” y el segundo en “1”, así sucesivamente.

## Tipos de operadores en Java

Una vez que tenemos variables, las podemos usar para crear y formar **expresiones** que **regresen valores**. Esto gracias a los operadores:

### Operadores aritméticos:

|  |  |  |
| --- | --- | --- |
| Operador | Nombre | Ejemplo |
| + | Adición | a+b |
| - | Sustracción | a-b |
| \* | Multiplicación | a\*b |
| / | División | a/b |
| % | Módulo | a%b |

### Operadores de asignación:

|  |  |  |
| --- | --- | --- |
| Operador | Aplicación | Desglose |
| += | a += b | a = a + b |
| -= | a -= b | a = a – b |
| \*= | a \*= b | a = a \* b |
| /= | a /= b | a = a / b |
| %= | a %= b | a = a % b |

### Operador incremento y decremento:

|  |  |  |  |
| --- | --- | --- | --- |
| Operador | Nombre | Ejemplo | Desglose |
| ++ | Incremento | i++ | i = i +1 |
| -- | Decremento | i-- | i = i-1 |

Se puede escribir en forma de prefijo y posfijo: ++i, i++

//al usar ++l

1. Incrementa el valor l + 1

2. Asignar el valor a l l = l

//al usar l++

1. Asignar el valor a l l = l

2. Incrementa el valor l + 1

### Operadores de equidad:

|  |  |  |
| --- | --- | --- |
| Operador | Nombre | Ejemplo |
| == | Igualdad | a == b |
| != | desigualdad | a != b |

Estos operadores funcionan como booleanos, nos permiten comprar si un valor es igual o diferente a otro, mostrando “true” o “false” respectivamente.

### Operadores relacionales:

|  |  |  |
| --- | --- | --- |
| Operador | Nombre | Ejemplo |
| < | Menor que | a < b |
| > | Mayor que | a > b |
| <= | Menor o igual que | a <= b |
| >= | Mayor o igual que | a >= b |

### Operadores lógicos:

|  |  |  |
| --- | --- | --- |
| Operador | Nombre | Ejemplo |
| && | AND | a && b |
| || | OR | a || b |
| ! | NOT | !a |

|  |  |  |
| --- | --- | --- |
| a | b | a || b |
| f | f | v |
| f | v | f |
| v | f | f |
| v | v | f |

### Tablas de verdad:

|  |  |  |
| --- | --- | --- |
| a | b | a && b |
| f | f | f |
| f | v | f |
| v | f | f |
| v | v | v |

|  |  |
| --- | --- |
| a | !a |
| v | f |
| f | v |

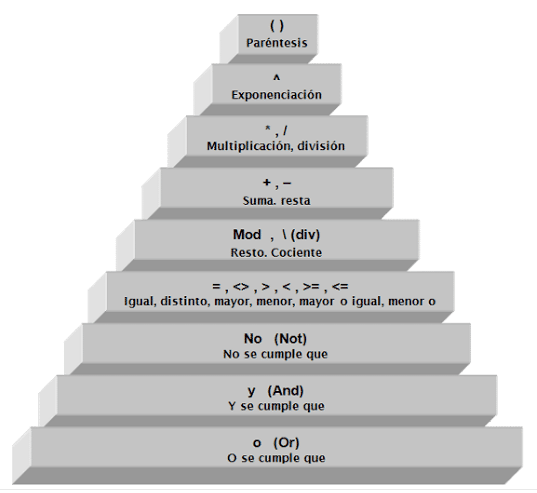
NOT

AND

OR

## Procedencia de operadores

Java tiene un mecanismo llamado Procedencia de Operadores, que permite jerarquizar las operaciones dándoles una prioridad con respecto a otra. La cual es muy similar a la jerarquización aritmética común, a continuación, se muestra un gráfico donde el punto más alto es la prioridad más alta.

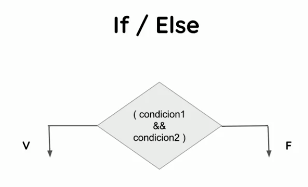


Como podemos observar, la *jerarquía regular de las reglas aritméticas* difiere un poco a la de *Java*, pues esta última cambia la prioridad de los paréntesis y la exponenciación. También es fácil notar que los operadores lógicos se toman en cuenta en este mecanismo de Procedencia de Operadores cuando en la jerarquía aritmética estos no figuran.

# Control de fujo en Java

Podemos controlar el flujo usando sentencias condicionales, ciclos, etc.

## If / Else



Lo que hace es colocar dentro de un cuadro condicional una de nuestras sentencias lógicas. Las compara y dependiendo del resultado tomará una u otra decisión. En el caso del diagrama que vemos en la parte superior, dependiendo si el resultado es verdadero tomará una decisión diferente a cuando obtenga un resultado falso.

**En código:**

if (condición){

instrucciones

} else {

Instrucciones

}

Entonces, una vez que tengamos escrita nuestra condición en caso de que esa condición se cumpla en una primera instancia, se ejecutarán las instrucciones contenidas en el “if”, en caso de que esa condición no se cumpla, realizará las instrucciones contenidas en “else”.

## Switch

Esta sentencia es similar a If/Else, también ejecuta un bloque de condición, con la diferencia de que en el switch vamos a tener una mayor cantidad de “caminos” que se pueden tomar.

**Por ejemplo:**

switch(a){

case valor1;

break;

case valor1;

break;

default;

Break;

}

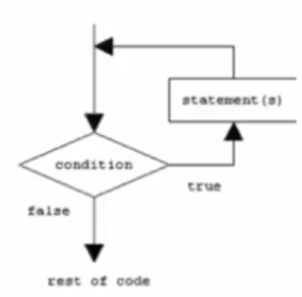
**Algunas reglas importantes a la hora de declarar switch**

* Los valores duplicados de los case no están permitidos.
* El valor para un **case** debe ser del mismo tipo de datos que la variable en el **switch.**
* El valor para un **case** debe ser una constante o un literal.
* Las variables no están permitidas.
* La declaración de **break** es opcional.
* La instrucción **default** es opcional y debe aparecer al final del **switch**.

## Ciclo while

Esta sentencia de control de flujo es útil cuando queremos que una condición se ejecute más de una vez o que se esté ejecutando repetidamente. Es parecido a if/else, pues también cuenta con solo dos opciones, pero con la diferencia de que cuando sea verdadero se ejecutarán las instrucciones y regresará al bloque de comparación y volverá a ejecutar la comparación con el resultado.

Cuando la condición resulta ser un false, saldrá de ese pequeño ciclo y continuará con el resto del código que le sucede.



Sintaxis:

while (condición) {

//instrucciones

}

Entonces, mientras la condición resulte verdadera, las instrucciones se seguirán ejecutando, pero si la condición resulta falsa, entonces saldrá del ciclo y continuará con el flujo del código

## Do while

Es muy similar a while, con la diferencia de que las instrucciones se ejecutan al menos una vez y después compara. Entonces, mientras que en while primero se deben cumplir con los requisitos para entrar en el bucle, con do while se puden empezar a generar las instrucciones del ciclo y después comenzamos a comparar que esos requisitos se cumplan.

En código;

do {

//instrucciones

} while (condición);

Es muy común utilizarla cuando trabajamos con menús.

## Ciclo for y foreach

### for

Este controlador de flujo es muy útil para reducir líneas de código, pues mientras que en el ciclo while tenemos que poner en líneas separadas la inicialización de la variable, la condición y el incremento que tendrá la variable, cuando usamos el ciclo for estas tres acciones se realizan en una sola línea, tal y como se muestra en la sintaxis de a continuación.

**Sintaxis:**

for (inicialización; condición; incremento){

//instrucciones

}

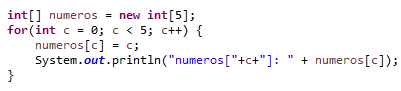
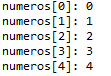
Para comprender mejor cada uno de estos conceptos, establezcamos que:

* **Inicialización:**
  + En esta parte es donde declaramos la variable. En el ciclo while teníamos que declarar en una línea aparte esta sentencia tal que así:
    - **int a = 1**; //en este caso la variable es un dato de tipo entero y tiene un valor de 1
* **Condición:**
  + Esta sentencia es la parte que iría como tal dentro de nuestro while, así:
    - while **(a <= 5)**
* **Incremento:**
  + Por último, el incremento es la línea de código en la que definimos el incremento de nuestra variable, lo cuál nos permitirá darle lógica a nuestro ciclo y que no se genere un ciclo infinito.
    - **a++**; //Esta línea de código contenida dentro del ciclo while permite darle un incremento a la variable, para que cada que haga la comparación sea un valor diferente al anterior.

### foreach

**Foreach** es una forma aún más **simplificada de ejecutar for**, con la diferencia de que ahora no tendremos acceso al índice de cada valor, sino que foreach solo muestra el dato que está dentro del arreglo.

Entonces, cuando usamos el foreach, ya no estaremos viendo la ubicación de cada dato con su índice (números[n], i.e.) sino que solo veremos el **valor** de cada dato, esta comparación se muestra a continuación:







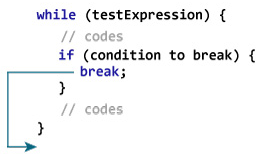
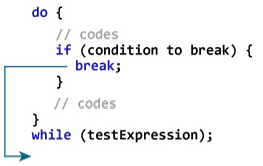
Por lo tanto, como se puede observar, el código que tenemos con foreach es mucho más corto, así como lo es la impresión en la consola (pues ahora no tenemos el índice de c antes de cada valor), por lo que, a modo de performance el controlador de ciclo foreach es más eficiente.

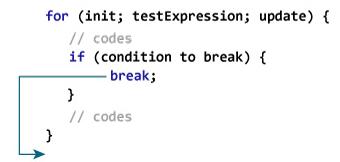
# Definiciónes de Break, Continue y Return

## Break

La sentencia break termina el bucle inmediatamente y el control del programa se mueve a la siguiente sentencia hallada después del bucle.

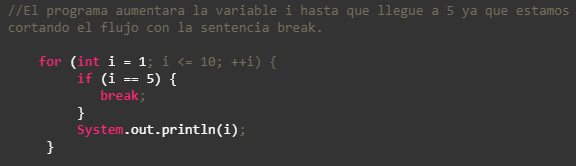
Nota: Normalmente se usa con declaraciones de toma de decisión también llamados controladores de flujo (if, if/else, while, do while, for). Veámoslo en un ejemplo de código:





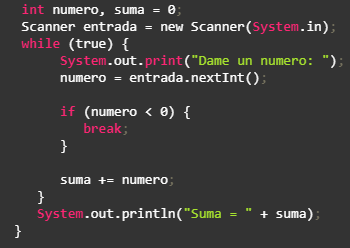
Entonces, lo que está ocurriendo en estos casos es que si la condición contenida dentro de los if es verdadera, atumáticamente se termina el cliclo y sale de él para continuar con el código.

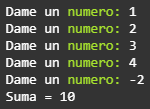
**Ejemplo de un break en un ciclo for:**





**Ejemplo de break en un ciclo while:**





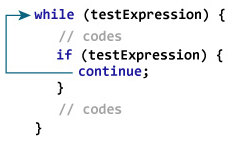
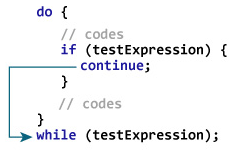
En este caso, el ciclo while se ejecuta hasta que el usuario introduce un número negativo. Si esto ocurre, la sentencia if se rompe y termina el bucle while.

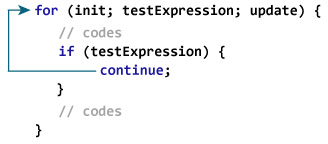
## Continue

La sentencia continue salta la iteración actual del bucle.

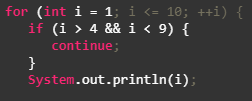
Cuando se ejecuta la sentencia continue, el control del programa salta al final del bucle.

Nota: Normalmente se usa con declaraciones de toma de decisión también llamados controladores de flujo (if, if/else, while, do while, for). Veámoslo en un ejemplo de código:



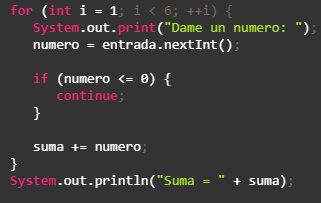
**Ejemplo de continue en un ciclo for:**

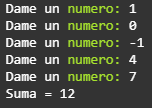




Aquí lo que está ocurriendo es que cuando el valor de i se convierte en más de 4 y menos de 9, se ejecutará la sentencia continue, lo que omite la ejecución de System.out.print.ln(i);

**Ejemplo 2:**



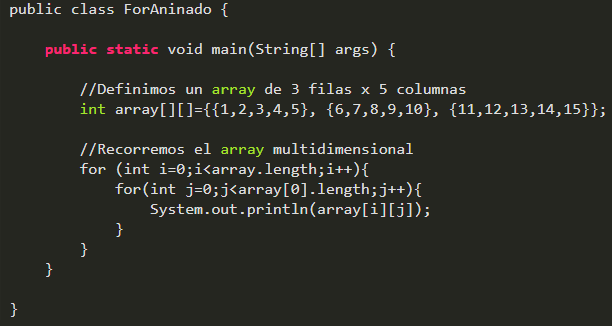


De igual forma, un programa más complejo sería igual que en el caso del ejemplo del break, pero ahora estableciendo un máximo de 5 números a sumar, si el usuario pone un número negativo o igual a 0, el programa lo saltará.

Ciclos for anidados

Además de manejar bucles for sencillos como en el ejemplo anteriro, podemos manejarlos también anidados como es el caso que quisiéramos iterar un array de 2 0 3 dimensiones.

La mejor forma de verlo en práctica es utilizando un arreglo de 2 dimensiones; haríamos lo siguiente:



Ahora, si lo hiciéramos con un foreach tendríamos lo siguiente:

