# 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

**Variable**: una variable es un espacio en memoria al que le asignamos un contenido; puede ser un valor numérico, 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 (-128 a 127)
  + short: 2 bytes (-32,768 a 32.767)
  + int: 4 bytes (-2,147,483,648 a 2,147,483,648)
  + long: 8 bytes (-9,223,372,036.854,775,808 a 9,223,372,036.854,775,808)
* **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, $ o \_:** 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:** Será el nombre nuestra clase y esto significa que siempre va 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 fácil 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 **abscisas** de nuestro plano cartesiano y los elementos contenidos en esas filas (**las columnas**), las **ordenadas** 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:

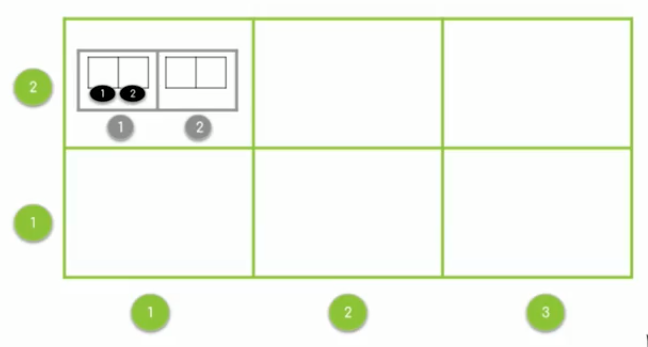


Ilustración : los arreglos de color gris (2D) deben estar también dentro de los demás arreglos en color verde (3D), hospedando la misma cantidad de arreglos en negro. De este modo, estamos hablando de un arreglo de n dimensiones.

## 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, que respectivamente 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 ++i

1. Incrementa el valor i + 1

2. Asignar el valor a i i = l

//al usar i++

1. Asignar el valor a i i = l

2. Incrementa el valor i + 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 |

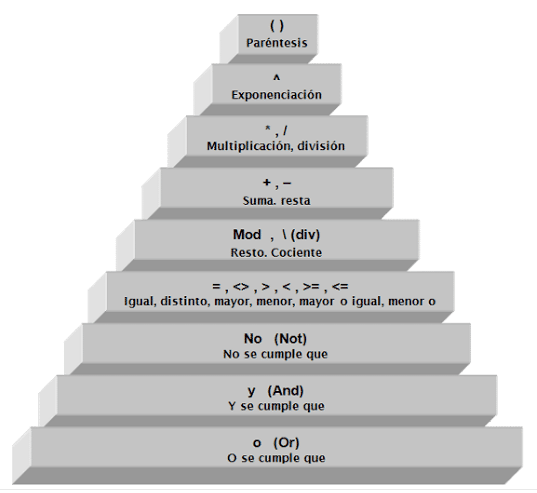
NOT

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.

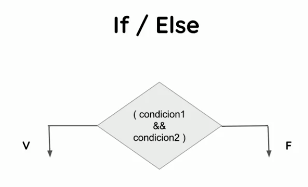


Como podemos observar, la *jerarquía regular de las reglas aritméticas* difiere un poco a la de *Java*, pues esta última cambia la prioridad de los paréntesis y la exponenciación. También es fácil notar que los operadores lógicos se toman en cuenta en este mecanismo de Procedencia de Operadores cuando 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;

}

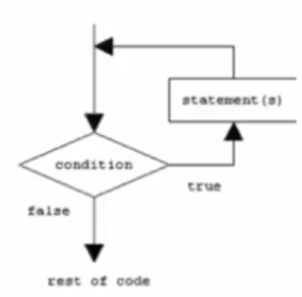
### Algunas reglas importantes a la hora de declarar switch

* 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

}

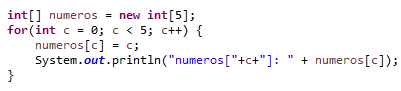
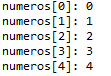
Para comprender mejor cada uno de estos conceptos, establezcamos que:

* **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 cual 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:







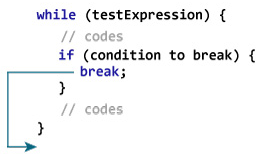
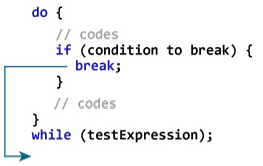
Por lo tanto, como se puede observar, el código que tenemos con **foreach es mucho más corto**, así como lo es la **impresión** en la consola (pues ahora no tenemos el índice de c antes de cada valor), por lo que, a modo de performance el controlador de ciclo foreach es más eficiente.

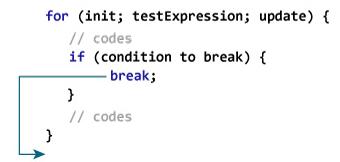
# Definiciónes de Break, Continue y Return

## Break

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

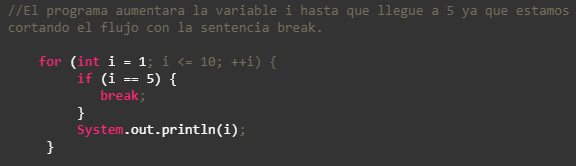
Nota: Normalmente se usa con declaraciones de toma de decisión también llamados controladores de flujo (if, if/else, while, do while, for). Veámoslo en un ejemplo de código:





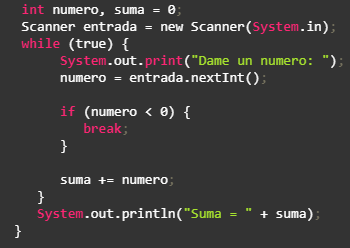
Entonces, lo que está ocurriendo en estos casos es que si la condición contenida dentro de los if es verdadera, atumáticamente se termina el cliclo y sale de él para continuar con el código.

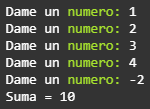
**Ejemplo de un break en un ciclo for:**





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





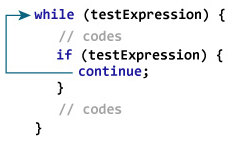
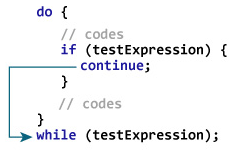
En este caso, el ciclo while se ejecuta hasta que el usuario introduce un número negativo. Si esto ocurre, la sentencia if se rompe y termina el bucle while.

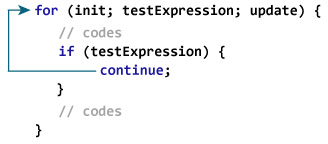
## Continue

La sentencia continue salta la iteración actual del bucle.

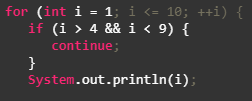
Cuando se ejecuta la sentencia continue, el control del programa salta al final del bucle.

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:



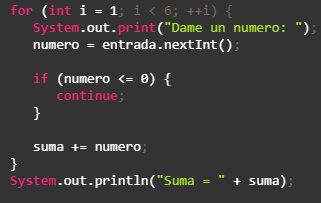
**Ejemplo de continue en un ciclo for:**

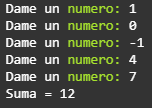




Aquí lo que está ocurriendo es que cuando el valor de i se convierte en más de 4 y menos de 9, se ejecutará la sentencia continue, lo que omite la ejecución de System.out.print.ln(i);

**Ejemplo 2:**



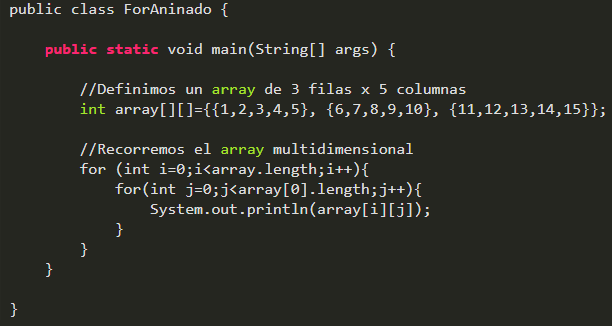


De igual forma, un programa más complejo sería igual que en el caso del ejemplo del break, pero ahora estableciendo un máximo de 5 números a sumar, si el usuario pone un número negativo o igual a 0, el programa lo saltará.

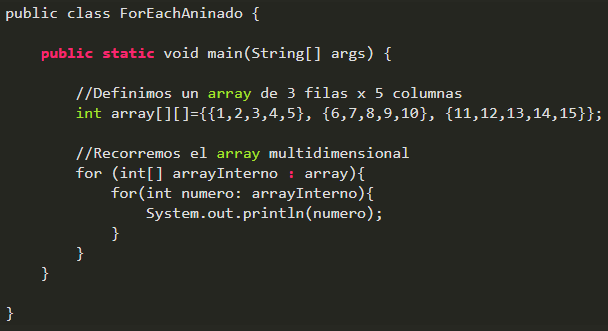
# Ciclos for anidados

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:



# Programación Orientada a Objetos

Para comprender la POO, debemos cambiar la forma de ver las cosas, pues la programación orientada a objetos es una forma mucho **más abstracta** de verlas.

Se trata de **descomponer un problema** en subproblemas y más subproblemas.

## Problem Domain

Un dominio de problema **no** es simplemente ver **rapidamente** el problema, sino **dividir** la información y **analizar por separado** varios subproblemas. Cuando llegamos a esto, estamos hablando de un *problem domain*, en donde **recopilamos los requisitos** del cliente y tenemos por escrito un **alcance** del proyecto; lo que queremos lograr.

Lo mejor es tener por escrito hasta dónde se quiere llegar con este programa, qué tanto vamos a **resolver el problema**, por esto mismo, es que es bueno dividir la información.

La programación orientada a objetos es ver el problema como un escenario y tratar de simularlo con varios objetos.

## Identificando objetos:

### Los objetos pueden ser **físicos o conceptuales**

* Físicos:
  + Es un objeto que es posible tocar, que es manipulable, tangible. Un usuario sería un buen ejemplo de un objeto físico.
* Conceptual:
  + Un objeto conceptual es algo intangible, algo que no podemos ver fisicamente. Una sesión de usuario sería un buen ejemplo de objeto conceptual.

### Los objetos tienen los siguientes atributos:

* tamaño
* nombre
* forma
* representan el estado del proyecto

### Los nombres y atributos de los objetos por lo general son sustantivos.

* user, session

Las operaciones suelen ser verbos o sustantivo y verbo, estas son todas las acciones que puede realizar un objeto.

* login, makeReport

## Análisis de un objeto y qué es una clase.

Una vez que comprendimos lo que es un objeto y a identificarlos, podemos empezar a analizarlos.

Usemos el ejemplo de un auto:

Un auto va a tener una *matricula, una marca y un modelo*. Todos estos son **sustantivos**, por lo tanto, sí, son **atributos del auto.**

Los objetos también tienen **comportamientos**, los comportamientos de un auto podrían ser:

El objeto auto **puede**: arrancar, ir de reversa e incluso frenar, podría tener muchos más, pero escencialmente, estas son todas las operaciones que puede realizar un auto.

Pero si nosotros necesitamos trabajar al objeto en **un escenario especifico**, asi es como hay que hacerlo:

Dependiendo el *Problem Domain* que manejemos, podemos **colocar a nuestro objeto** en ese escenario específico.

Continuando con el ejemplo del auto; lo colocaremos en un **escenario** en donde una c**oncesionaria de autos** está vendiendo este objeto “auto”.

Entonces, los atributos y comportamientos podrían ser distintos, cambiar algunos de ellos o incluso agregar otros adicionales.

Si el auto está dentro de un sistema en donde ha de ser vendido, contaríamos con los siguientes atributos:

* matrícula.
* Marca.
* Modelo.
* **Precio**.
* **Vendido**

Como podemos ver, ahora dentro de los atributos del objeto contamos con “precio” y con un estado de “vendido”

Así mismo, los comportamientos del objeto dentro de este escenario podrían ser:

* mostrarDatos.
* esVendido.

Por la tanto, como podemos ver, las operaciones del auto son completamente diferentes, pues ahora ya no interesa si el auto puede arrancar, frenar, reversa, etc. Ahora solo me interesa que el objeto pueda mostrarDatos, lo que mostraría los atributos del objeto, para después mostrar también el estado “esVendido” del auto, en donde, si el atributo “vendido” es verdadero, podríamos ver si este ha sido vendido o aún no.

## Clase

Una clase es la forma en cómo **definimos el objeto** y la clase también sirve para generar más objetos.

Las clases son descriptivas, o sea que son **plantillas**.

Por ejemplo, en caso de tener más de un auto, vamos a abstaer todos los atributos que posiblemente puedan tener. Entonces, una clase tendría los siguientes atributos y comportamientos:

* Atributos:
  + Id.
  + Matrícula.
  + Marca.
  + Año.
  + Precio.
  + Vendido.
* Comportamientos
  + mostrarDatos.
  + esVendido.

Y así es como estructuramos una clase en Java:

Public class Auto{

atributos

int id;

String matrícula

String marca

String modelo

double precio

boolean vendido

comportamientos

public void mostrarDatos() {

}

public boolean es Vendido() {

}

}

Esta estructura es muy similar a una **receta de cocina**, primero colocaremos todos los *“ingredientes”* que serán nuestros *atributos* y después colocaremos las “instrucciones” que serán los comportamientos, estos indican lo que podemos hacer con esos *“ingredientes”.*

## La sentencia import

import define a las clases o grupos de **clases externas** que utilicemos en la clase, esta es una palabra reservada y la veremos siempre en el encabezado del archivo de la clase, por ejemplo:

import java.util.Date;

Podemos importar tan **solo una clase** como en el ejemplo anterior o un paquete de clases:

import java.util.\*;

Además de que es posible incluir **múltiples sentencias** import:

import java.util.Date;

import java.util.Calendar;

Es una buena práctica **especificar la importación** completa del paquete de clases\*, sin embargo **puede causar conflictos** con las clases que vivan en diferentes paquetes pero tengan el **mismo nombre**, por ejemplo:

java.util.Date

java.sql.Date

Cuando tengamos algo **similar** podemos usar la siquiente sintaxix para diferenciar.

Elegimos qué import **conservar**:

import java.util.Date;

Y la **otra clase** la llamamos de la siguiente forma:

java.sql.Date date = new java.sql.Date();

En el caso de **miembros static**, que veremos más adelante, se importarán de la siguiente forma:

import static java.lang.Math.PI; // importa solo la variable estática PI

import static java.lang.Math.\*; // importa solo los elementos estáticos en esa clase

Por lo tanto, podemos usarlos asi:

double r = cos(PI \* theta);

Nota: Por default siempre se importa java.lang\*. No es necesario importar las clases que estén en el mismo paquete. Import será necesario en todo momento para poder crear funciones con operaciones matemáticas complejas o poder establecer la fecha exacta a mostrar.

## Métodos y método Constructor en Java

### Métodos

En java las funcionalidades y comportamientos de un objeto se conocen como “métodos” y un método tiene la siguiente estructura:

public int suma (int a, int b){

}

Modificador de acceso:

Valor de retorno: o sea que, si es una suma, devolverá el resultado. Si los valores que estamos sumando son enteros, entonces obviamente el resultado también debe ser un entero.

Nombre: colocamos en este espacio el nombre del método. En este caso usaremos lower Camel Case, empezaremos siempre con una minúscula y si el nombre del método se compone por dos o más palabras, las iniciales las pondremos con mayúsculas.

Argumentos: estos serán los datos que entren al método. Estos deben estar separados por una coma, y también pude no haber ningún argumento “()”.

Los métodos tienen un valor de retorno explícitamente invocado en su cuerpo. Si el método no tiene un retorno, debe colocarse la palabra void en el área de “valor de retorno”

### Declarando un objeto.

Cuando queremos que la clase **cobre vida,** esto quiere decir que la utilizaremos **a partir de un objeto**, o sea, que al **declarar un objeto** tenemos que colocar el **tipo de objeto**, esto estará dado por la clase de donde queremos generar un objeto, después vendrá el nombre del objeto, así:



Pero sucede lo mismo que con los arreglos, si sólo lo declaramos y no le hemos dado “vida”, no lo podremos utilizar. Para hacer esto, utilizaremos la siguiente sintaxis:



Por lo tanto, una vez que hayamos escrito de esta forma el nombre del objeto, eso significará que estamos **instanciando un objeto** y cuando decimos que estamos **generando una instancia** a partir de una clase, esto significará que estamos generando un **objeto** a partir de una **clase**. Entonces, una vez que nosotros hemos creado el objeto a partir del **método constructor** con la sintaxis mencionada, quiere decir que ese objeto ya “existe”, ya está ocupando un espacio de memoria dentro de nuestro computador.

Si queremos instanciar y declarar un objeto en una sola línea, podemos hacerlo así:



### Método constructor

El método constructor es diferente a los métodos ya vistos. Este método es el responsable de **crear nuevas instancias** de una clase.

Tiene el **mismo nombre** que la clase que inicializa.

Además, hace uso de la palabra reservada “new” para invocarlo, también utiliza **cero o más argumentos** contenidos dentro de los paréntesis que siguen al nombre, al igual que el resto de los métodos.

El método constructor **nunca regresará un valor** esta es, junto con el nombre, una de las principales diferencias entre el método constructor y los otros.

Este es provisto por java por default, ya está definido para crear un objeto.

## Miembros estáticos y final

### Método estático:

Como ya sabemos, la forma clásica de acceder a los métodos es a través del nombre del método “.” el nombre del método.

Estos se pueden usar en toda la clase, además de poderse utilizar en otras clases. Está definido por la palabra reservada **static.**

Otra forma de ser accesado **es indicando el nombre de la clase, la notación punto y el nombre del método**, así:

Math.random();

Math.sqrt(25);

Math.PI :

En este caso Math es la clase y random, sqr y PI son los métodos a utilizar.

Otro ejemplo a continuación.

public class Calcularora{

public static int suma(int a, int b){

return a+b;

}

}

Entonces, para llamar al método suma solo es necesario llamarlo de la siguiente manera:

Calculadora.suma(5, 2);

Este método puede ser invocado en una clase que **no tiene instancias de la clase**. Por lo tanto, tanto los métodos como las variables static nos permiten **ahorrar memoria**.

El **metodo main** siempre es un método estático, ya que, cuando se invoca no existen objetos creado con anterioridad porque la ejecucjión del programa aún no ha comenzado.

El tipo main **siempre será void** (nulo), ya que **no** es un tipo función que devuelva un valor. Su misión es arrancar la ejecución, no devolver un valor.

Un método de clase o static es aquel que *puede ser invocado sin existir una instancia*

#### ¿Por qué todos los metodos de main deben ser estáticos?

El hecho de sean estáticos quiere decir que sólo pueden referenciar varialbes locales, otros métodos estáticos, miembros de mainclass u otras clases que también son estátios.

Un método static NO puede acceder a una variable no-static, porque no es una instatncia, el método static no sabe nada sobre ningun objeto instanciado.

La equivalencia es:

* En static = clase
* No-static = instanciada

Haciendo el método llamado por la JVM(main()), un método static significa que la JVM no tiene que crear una instancia de nuestra clase solo para inciar el código.

### Sobrecarga de métodos

A veces necesitamos que dos o más metodos tengan el **mismo nombre**, pero con **diferentes argumentos.**

public class Calculadora{

public int suma(int a, int b){

return a+b;

}

public float suma (float a, float b){

return a+b;

}

public float suma (int a, float b){

}

}

De este modo podemos hacer uso de una sola clase, para manejar distintos tipos de datos sin generar ningúna especie de conflicto.

La sobrecarga de constructores se usa parar **inicializar objetos**.

public class auto {

int id;

String matrícula

String marca

double precio

boolean vendido

public auto(){

}

public auto (int id, String matricula, String precio){

this.id = id;

this.matricula = matricula;

this.precio = precio;

}

}

Modificadores de acceso

Los modificadores de acceso son útiles para poder trabajar con clases que no se puedan modificar constantemente por el usuario, estos tienen un cierto nivel de acceso a estas modificadores. Estos niveles y de cada uno de estos modificadores, son:

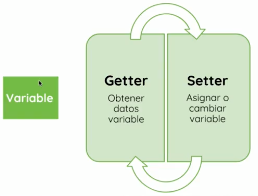
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modificador | Clase | Package | Subclase | Todos |
| Public | Si | Si | Si | Si |
| Protected | Si | Si | si | No |
| Default | Si | Si | No | No |
| Private | Si | No | No | No |

Cuando “escondemos” alguno de los atributos cambiando su modificador de acceso, lo que hacemos es una **técnica** conocida como **encampsulamiento**, entonces cuando la encapsulamos al nivel de la clase, estamos logrando que nuestros atributos de objetos no sean modificados desde algúna clase externa.

## Getters y Setters

Estos metodos nos ayudan a leer y escribir especificamente los valores de las variables miembro.

Estos funcionan como un pequeño ciclo con las variables, así:



Con getter estaríamos **obteniendo** datos de la variable y con el setter, lo que hacemos es **asignarle** un valor a la variable.

#### Diferencias entre variables y objetos:

Las **variables** son entidades elementales (**muy sencillas**) mientras que los **objetos** son entidades **complejas** que pueden estar formados por la **agrupación** de muchas **variables y métodos**.

Las variables **primitivas** suelen vivir en la **memoria stack,** pues su tiempo de vida suele ser muy corto, estas variables son las que están definidas de un metodo, mentras que las variables **objeto** viven en la **memoria heap.**

Int i = 0; es una variable y se almacena en stack

Auto miAuto = new Auto(); es un objeto, por lo que es más complejo y requiere mayor cantidad de memoria.

## Qué es la herencia?

En algunas circunstancias, es necesario **utilizar el estado y comportamiento** de una clase en conjunto con **otras clases**.

Estas herencias son muy similares a las herencias biológicas que experimentamos nosotros, pues puede suceder más de una vez y secuencialmente, el padre de una clase le hereda atributos a su clase hijo, y esta a su vez se las heredará a sus hijos.

Se establece una **relación padre-hijo** entre dos objetos diferentes.

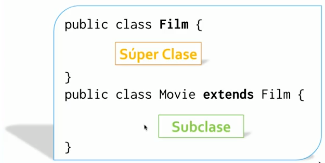
La idea de la herencia es permitir la creación de **nuevas clases** basadas en **clases existentes**.

#### Super y this.

Cuando comenzamos a trazar el código aparecerán las palabras super y this. Estas palabras permiten relacionar entre clases los atributos a heredar.

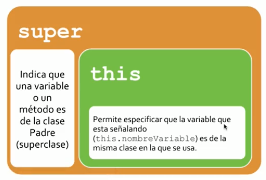
La clase padre, la cual es la que contiene todos esos atributos que las clases hijo es conocida como súperclase. Entonces, cada que encontremos la palabra súper dentro del código, estará haciendo referencia a la clase padre.

Cuando apliquemos la herencia, la palabra clave que define la herencia entre una clase y otra es la palabra **extends**, o sea, que esta clase “extienede” los atributos de la súperclase, y ahora **todos los atributos** de esa clase padre, corresponderán tambien a la subclase.



Una subclase hereda **todos los miembros de su súperclase** que están declarados como **public o protected.**

Entonces, a continuación se muestra la forma en la que **super y this** se relacionan con las **herencias:**



## Polimofrismo

Si tenemos un un **método o un comportamiento** de una clase padre y ahora **al momento de sobreescribirlo** le estamos dando otro comportamiento, estamos hablando de *poliformismo*.

El polimorfismo nos da la posibilidad de construir varios métodos con el mismo nombre, pero **diferente comportamiento** (siempre y cuando esté aplicandose la herencia)

### Sobreescritura:

Cuando una clase hereda de otra, y en esta **clase hija se redefine un método** con una implementación distinta a la de la clase padre.

Los métodos marcados como final o static NO se pueden sobreescribir.

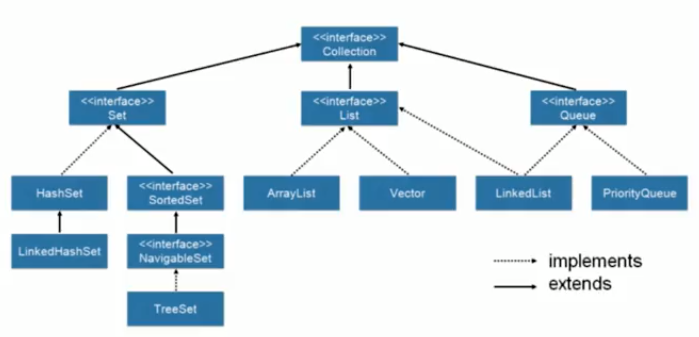
Un **costructor** es una subclase usando los miembros heredados de la superclase, **con argumantos diferentes.**

## Interfaces

Es un tipo de referencia similar a una clase que podría contener solo constantes, definiciones de métodosm métodos con modificador de acceso default.

Colecciones de datos

Se diferencían de los arrays en que su tamañno no es fijo, son dinámicos y se pueden realizar operaciones de añadir, eliminar, obtener, encontrar o recorrer una colección.



### List

Es una interfaz que se extiende de la interfaz Collection, se utiliza para almacenar colecciones de objetos del paquete java.util

La diferencia entre **set** y **list** es que las colecciones los elementos van a estar ordenados y además tendremos la posibilidad de que haya objetos repetidos. En el caso de **set** puede estar o no ordenada y no es posible repetir elementos.

List, además, cuenta con varios métodos bastante útiles:

* **add(Object o):** esto añade un objeto al final de la lista. O sea, si ya tenemos una lista de objetos definida, este método va añadiendo los elementos que requiramos sin alterar el orden, añadiéndolos hasta el final
* **add(int índice, Object o):** con este método (el cual, podemos observar que estamos haciendo uso de la sobrecarga de métodos) nos es posible añadir un objeto a la lista en la posición indicada por “índice”, en caso de que no queramos que se añada hasta el final de la lista.
* **get(int índice):** una vez que ya tenemos una colección de datos podemos ir obteniendo un dato específico a partir de un índice, esto lo logramos con este método.
* **remove(int índice):** al mismo tiempo, una vez contemos con una cierta colección de datos, es posible retirar un objeto de nuestra lista, gracias al método remove a partir de un índice indicado.
* **clear():** esto nos permite eliminar todos los objetos de la lista
* **indexOf(Object o):** devuelve la posición de la primera vez que el elemento coincida con el objeto pasado por parámetro. Si el elemento no se encuentra devuelve -1.
* **lastIndexOf(Object o):** este método devuelve la posición de la ultima vez que un elemento coincida con el objeto pasado por parámetro. Si el elemento no se encuentra devuelve -1.
* **size():** devuelve el número de elementos de la lista.
* **contains(Object o):** este método verifica que un objeto se encuentre en la lista. Devuelve verdadero si en la lista aparece el objeto pasado por parámetro, paralo cual utiliza intrínsecamente el método equals().

#### ArrayList

Almacena un arreglo de objetos que **pueden cambiar de tamaño**, tiene una capacidad que crece automáticamente.

Sintaxis:

ArrayList<String> androids = new ArrayList<String>();

androids.add(“Jelly Bean”);

androids.add(“Kit Kat”);

androids.add(“Lollipop”);

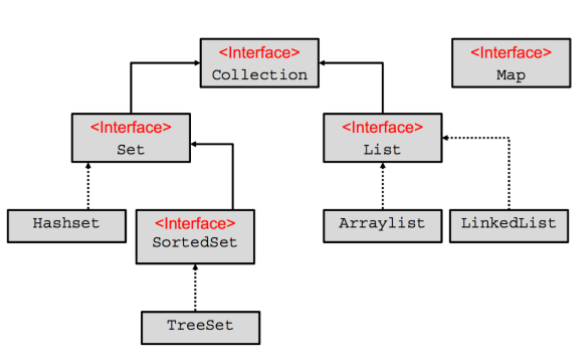
#### Vector

Un vector es similar a un array, con la diferencia de que un vector usa **hilos** y **está sincronizado**, mientras que un array no.

### Más colecciones

Entonces, como ya vimos, las colecciones nos ayudan a generar listas de objetos, únicamente objetos, para definir el tipo de objetos que almacenará la colección usamos la sintaxis diamante <>.

En java existen colecciones que podemos usar para muchos tipos de casos, comenzaremos aprendiendo su árbol familiar:



Como podemos observar, el elemento más alto es la interfaz **Collection**, para lo cual, partiendo de su naturalidad de interface, entendemos que tiene una serie de métodos “básicos” donde su comportamiento será definido a medida que se vaya implementando en más elementos. De ella se desprenden principalmente las interfaces **Set** y **List.**

La interface **Set** tendrá las siguientes características:

* almacena objetos únicos, no repetidos.
* La mayoría de las veces los objetos se almacenarán en desorden.
* No tenemos índice.

La interface **List** tiene estas características:

* Puede almacenar objetos repetidos.
* Los objetos se almacenan en orden secuencial.
* Tenemos acceso al índice.

Si seguimos analizando las familias tenemos que de **Set** se desprenden:

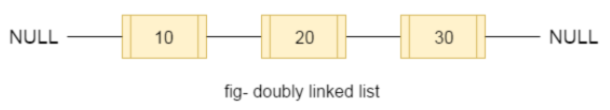
* Clase HashSet
* Interfaz SortedSet y de ella, la clase TreeSet.

**HashShet**: Los elementos se guardan **en desorden** y gracias al mecanismo llamado hashing (obtiene un identificador del objeto) permite **almacenar objetos únicos**.

**TreeSet**: Almacena **objetos únicos** y gracias a su estructura de árbol, el acceso es sumamente **rápido.**

Ahora, si analizamos la familia **List**, de ella se desprenden:

* Clase **ArrayList** puede tener duplicados, no está sincronizada, por lo tanto, es más rápida
* Clase **Vector** es sincronizada, los datos están más seguros, pero es más lento.
* Clase **LinkedList**, puede contener elementos duplicados, no está sincronizada (es más rápida). Al ser una estructura de datos doblemente ligada, podemos añadir datos por encima de la pila o por debajo



#### Map

Y ahora, hay un elemento que pareciera no estar incluido en la familia (**interfaz Map**). Esta tendrá como naturaleza, contar con los componentes key, value, donde el Value será el objeto insertado y el Key será el valor clave para obtenerlo.

Map también cuenta con sus propias ramas que se desprenden de él y son:

* **HasMap:** No permite ordenar sus elementos y es bastante eficiente a la hora de almacenar y obtener información
* **LinkedMap:**
  + Ordenado por inserción.
  + Permite ordenar por uso.
  + Eficiente lectura.
  + Poca eficiencia en escritura.
* **TreeMap:**
  + Ordenado por claves
  + Poco eficiente en todas sus operaciones
* **EnumMap:**
  + Permite enum como claves
  + Muy eficiente
* **WeakHashMap:**
  + Utilizado para crear elementos y que el sistema los vaya eliminando si es que no se utilizan.
  + Muy poco eficiente
* **Hash table:**
  + Considerado obsoleto
  + Utilizado en operaciones de concurrencia
* **ConcurrentHashMap:**
  + Utilizado en concurrencia
  + No permite nulos

## Streams en Java

Cuando nosotros utilizamos la clase System.in, nosotros estamos ingresando únicamente bytes, por lo que aún no tenemos texto completo. Después de esto, la clase InputStreamReader comienza a descifrar uno a uno todo ese flujo de bytes.

Existe otra clase que también es muy interesante es BufferedReader, lo cual hacer que InputStreamReader descifre estos bytes mucho más rápido.

Ahora, si queremos trabajar con salida de datos a partir de Java, lo que haremos es partir de una cadena de texto (objeto de tipo String). Lo siguiente es generar un archivo ficticio, un objeto de tipo file que simulará un archivo físico que será escrito en nuestro disco duro.

Entonces, continuamos haciendo uso de la clase FileOutputStream, la cual recibe el objeto file y lo convierte en un flujo de bytes.

Utilizando el objeto OutputStream podremos comenzar a controlar el contenido de la cadena y ahora la frase volverá a ser un flujo de bytes y junto con la clase BufferedWriter va a hacerlo mucho más rápido.