PAD

JAVA - PADSA

Fundamentos de JAVA

Recopilación de los fundamentos para programar en java 8+.

Convención de Nombres en Java

Una convención de nombres es un patrón que deben seguir los nombres de las variables para que el código esté organizado, entendible y sin repetidos.

- Java es sensible a mayúsculas y minúsculas, este punto es clave al seguir una convención.
- Las variables siempre deben comenzar con un simbolo de letra, `$` o `_`.
- No puedes usar el simbolo `-` en ninguna parte de la variable.

Las variables constantes son variables cuyo valor nunca va a cambiar, por lo que se deben escribir completamente en mayúsculas y usando el cáracter `_`.

Técnica de Naming: Camel Case

Camel Case es una convención muy popular para nombrar nuestras variables. Podemos usarlo en modo *Upper Camel Case* o *Lower Camel Case*, la diferencia es si comenzamos el nombre de la variable con mayúscula o minúscula.

```
// Upper Camel Case:
class SoyUnaClase {};

// Lower Camel Case
int soyUnNumeroInt = 10;
```

Debemos usar *Upper Camel Case* en los nombres de las clases y archivos. Y *Lower Camel Case* en los nombres de las variables o métodos.

Tipos de datos numéricos

Tipos de datos para números enteros (sin decimales):

- `byte`: Ocupa 1 byte de memoria y su rango es de -128 hasta 127.
- `short`: Ocupa 2 bytes de memoria y su rango es de -32,768 hasta 32,727.
- `int`: Ocupa 4 bytes de memoria y su rango es de -2,147,483,648 hasta 2,147,483,647. Es muy cómodo de usar, ya que no es tan pequeño para que no quepan nuestros números ni tan grande como para desperdiciar mucha memoria. Puede almacenar hasta 10 dígitos.
- `long`: Ocupa 8 bytes de memoria y su rango es de -9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807. Para diferenciarlo de un tipo de dato long debemos terminar el número con la letra `L`.

Por ejemplo:

```
// Int:
int n = 1234567890;

// Long:
long nL = 123456789012345L;
```

Tipos de datos para números flotantes (con decimales):

- `float`: Ocupan 4 bytes de memoria y su rango es de 1.40129846432481707e-45 hasta 3.40282346638528860e+38. Así como `long`, debemos colocar una letra `F` al final.
- `double`: Ocupan 8 bytes de memoria y su rango es de 4.94065645841246544e-324d hasta 1.79769313486231570e+308d.

Por ejemplo:

```
// Double:
double nD = 123.456123456;

// Float
float nF = 123.456F;
```

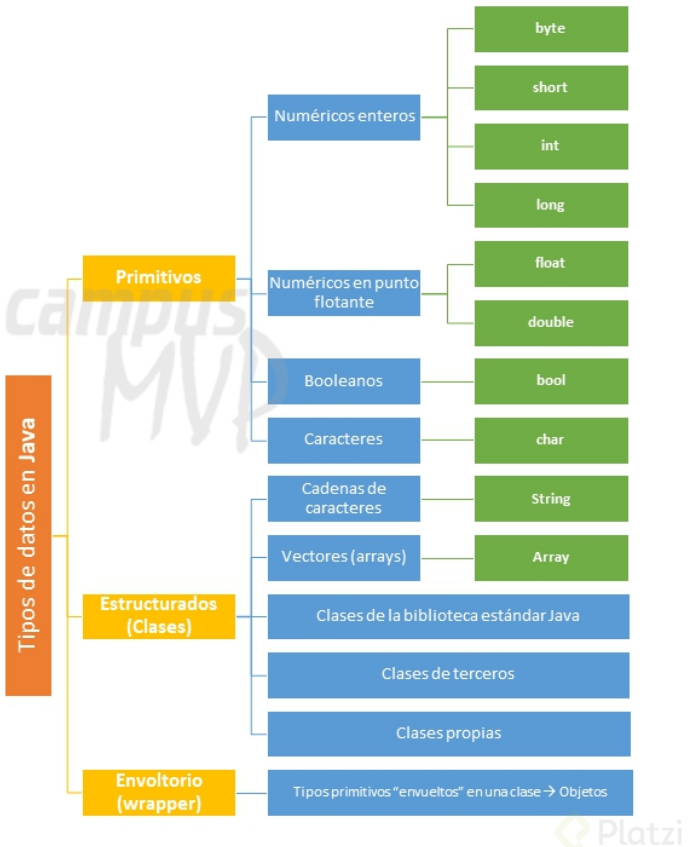
Tipos de datos char y boolean

- `char`: Ocupa 2 bytes y solo puede almacenar 1 dígito, debemos usar comillas simples en vez de comillas dobles.
- `boolean`: Son un tipo de dato lógico, solo aceptan los valores `true` y `false`. También ocupa 2 bytes y almacena únicamente 1 dígito.

Seguro te diste cuenta que siempre debemos escribir el tipo de dato de nuestras variables antes de definir su nombre y valor. Pero esto cambia a partir de Java 10: solo debemos escribir la palabra reservada `var` y Java definirá el tipo de dato de nuestras variables automáticamente:

```
var salary = 1000; // INT
var pension = salary * 0.03; // DOUBLE
var totalSalary = salary - pension; // DOUBLE
```

Recuerda que esto solo funciona con versiones superiores a Java 10.



Operadores de Asignación, Incremento y Decremento

Operadores de asignación:

- `+=`: `a += b` es equivalente a `a = a + b`.
- `=`: `a -= b` es equivalente a `a = a - b`.
- `=`: `a *= b` es equivalente a `a = a * b`.
- `/=`: `a /= b` es equivalente a `a = a / b`.
- `%=`: `a %= b` es equivalente a `a = a % b`.

Operadores de incremento:

- `++`: `i++` es equivalente a `i = i + 1`.
- `-`: `i--` es equivalente a `i = i - 1`.

Podemos usar estos operadores de forma prefija (`++i`) o postfija (`i++`). La diferencia está en qué operación se ejecuta primero:

Operaciones matemáticas

`Math` es una clase de Java que nos ayuda a ejecutar diferentes operaciones matemáticas:

```
Math.PI // 3.141592653589793
Math.E // 2.718281828459045

Math.ceil(2.1) // 3.0 (redondear hacia arriba)
Math.floar(2.1) // 2.0 (redondear hacia abajo)

Math.pow(2, 3) // 8.0 (número elevado a una potencia)
Math.sqrt(3) // 1.73... (raíz cuadrada)

Math.max(2, 3) // 3.0 (el número más grande)

// Área de un círculo (PI * r^2):
Math.PI * Math.pow(r, 2)

// Área de una esfera (4 * PI * r^2):
4 * Math.PI * Math.pow(r, 2)

// Volumen de una esfera ( (4/3) * PI * r^3):
(4/3) * Math.PI * Math.pow(r, 3)
```

Cast en variables: Estimación y Exactitud

En la programación hay situaciones donde necesitamos cambiar el tipo de dato de nuestras variables, esto lo conocemos como **Cast**.

Estimación:

```
double monthlyDogs = dogsQuantity / 12.0;
// monthlyDogs: 2.5 (pero no es posible, ¡no rescatamos medio perrito!)

int estimatedMonthlyDogs = (int) monthlyDogs;
// estimatedMonthlyDogs: 2

// Recuerda que el casteo no redondea, solo quita los decimales:
Math.sqrt(3) // 1.7320508075688772
(int) Math.sqrt(3) // 1
```

Exactitud:

```
int a = 30;
int b = 12;

a / b // = 2
(double) a / b // = 2.5
```

Casteo entre tipos de datos

Java nos ayuda a realizar casteo automático de los tipos de datos más chicos a otros más grandes.

Sin embargo, en algunos casos vamos a necesitar realizar un cast manualmente, así como aprendimos en la clase anterior (`(dataType) variableOperación`).

Por ejemplo: supongamos que declaramos dos variables `a` y `b` de tipo `int` y una variable `c` de tipo `double` que es igual a la división de las primeras dos variables.

En este caso, aunque definimos que el tipo de dato de `c` es `double`, Java automáticamente convertirá el resultado de la división a tipo `int`, ya que este es el tipo de datos de las dos variables que dividimos, pero seguirá respetando que la variable `c` es de tipo `double` y añadirá un decimal al final (`.0`).

Esto significa que muchas de nuestras operaciones pueden verse afectadas. Por ejemplo:

```
int a = 30;
int b = 12;

double c = a / b;
System.out.println(c); // 2.0 (??)
```

En este caso, ya que Java convierte nuestras variables automáticamente, debemos indicarle a nuestra variable `c` (de tipo `double`) que debe hacer cast de su valor para que Java no altere los valores de las variables y el resultado de la operación sea correcto:

```
int a = 30;
int b = 12;

double c = (double) a / b;
System.out.println(c); // 2.5
```

Es decir, como `a` y `b` son de tipo `int`, el resultado de una operación entre ambas variables será de tipo `int`, por lo que no tendrá decimales, pero si guardamos el resultado de esta división en una variable de tipo `double` añadiremos un `.0`.

Esto podemos solucionarlo si indicamos que además de que la variable `c` es de tipo `double`, el valor de esta variable también debe ser de tipo `double`. Esto significa que Java ejecutará la división entre `a` y `b` como si fueran de tipo `double`, por lo que tendrán decimales a pesar de haber sido definidas inicialmente como números enteros.

Archivos .JAR

Los archivos JAR (*Java Archive*) son archivos de Java con el código compilado de los archivos `.class` y comprimido con el formato ZIP para que más adelante sean interpretados y ejecutados por la máquina virtual de Java (JVM).

Para generar estos archivos podemos entrar a `File > Project Structure > Artifacts` y seleccionar la opción de `JAR > From modules with dependencies`. Luego de esto podemos compilar nuestro proyecto desde `Build > Build Artifacts > Build` y podremos nuestros archivos ejecutables en la carpeta `out/artifacts/`.

Podemos lanzar la aplicación desde una terminal de texto mediante:

java -jar aplic.jar

Sentencia if

```
boolean isBluetoothEnabled = true; // también podría ser false
int filesSended = 3;
if(isBluetoothEnabled) {
  fileSended++;
  System.out.println("Archivo enviado");
}
```

Alcance de las variables y Sentencia ELSE

- *Mientras más crecen nuestros programas, más lógica, complejidad y niveles añadimos. Estos niveles son el alcance que tienen nuestras variables, es decir, los lugares dónde pueden ejecutarse o no.

Estos niveles (en parte) son representados por las llaves (`{ ... }`) que envuelven nuestro código. Por lo tanto, entre más llaves envuelvan nuestro código, estaremos más niveles dentro y el alcance de las variables que definimos será un poco más limitado.

Solo podemos usar una variable si la definimos antes, en el mismo nivel o alguno anterior. Pero si declaramos una variable en un nivel posterior al resto de nuestro código, no podremos modificarla a menos que el código esté en su mismo nivel.

Por ejemplo:

```
// Primer nivel:
boolean condicion =true;
int numero1 = 1;

// Segundo nivel:
if (condicion) {
  // podemos modificar variables del primer nivel,
  // incluso desde el segundo nivel:
  numero1++;

  // También podemos crear y modificar
  // nuevas variables en este nivel:
int numero2 = 10;
  numero2++;
}

// Si volvemos al primer nivel, podemos seguir usando
// y modificando las primeras variables:
numero1--;
```

```
// Pero si salimos del segundo nivel no podemos volver a acceder
// a las variables que creamos allí:
System.out.println(numero2); // ERROR!
```

La sentencia `ELSE` es todo lo contrario a la sentencia `IF`: en vez de ejecutar una parte del código si la condición es verdadera, solo lo hará si la condición NO se cumple:

```
boolean isBluetoothEnabled = false;
int filesSended = 3;

if (isBluetoothEnabled) {
  fileSended++;
  System.out.println("Archivo enviado");
}else {
  System.out.println("El Bluetooth no está activado");
}
```**

Operadores Lógicos y Expresiones booleanas

Nuestros condicionales no solo pueden evaluar variables booleanas, también pueden evaluar si el resultado de una operación es verdadero o falso.

Por ejemplo:

```
boolean condicionA =true; // verdadero
boolean condicionB =false; // falso

boolean condicionC = 2 + 2 == 4; // verdadero
boolean condicionD = 2 + 2 == 5; // falso

boolean condicionE = "Pepito" == "Pepito"; // verdadero
boolean condicionF = "Pepito" == "Pepe"; // falso
```

Para esto debemos usar los operadores lógicos:

Operadores de equidad:

- **Igualdad**: `==`
- **Desigualdad**: `!=`

Operadores Relacionales:

- **Menor que**: `<`
- **Mayor que**: `>`
- **Menor o igual que**: `<=`
- **Mayor o igual que**: `>=`

Operadores lógicos:

- **&&**: AND (evaluar si dos o más condiciones son verdaderas).
- **||**: OR (evaluar si al menos una de las condiciones es verdadera).
- **!**: NOT (evaluar que la condición NO sea verdadera).

Recuerda que además de las sentencias `IF` y `ELSE`, también podemos usar `ELSE IF`. Esta la usamos cuando queremos evaluar alguna condición diferente a la condición del `IF` pero no exactamente al revés.

Por ejemplo:

```
if (noHayInternet) {
  System.out.println("No hay Internet");
}elseif (hayInternetPeroMuyPoquito) {
  System.out.println("Tienes muy poquito Internet");
}elseif (hayBuenInternetPeroNoEsSuficiente) {
  System.out.println("Casi casi, falta solo un poquito más");
}else {
  System.out.println("¡Tienes suficiente Internet!");
}
```

Sentencia Switch

La sentencia `Switch` nos ayuda a tomar decisiones con base en una o más condiciones, pero funciona un poco diferente:

Switch hasta Java 11:

```
switch (profe) {
case "Anahí":
    System.out.println("¡Profesora de Java!");
break;
case "Oscar":
    System.out.println("¡Profesor de React.js!");
break;
case "JuanDC":
```

```
    System.out.println("Oye niño, ¿qué haces aquí?");
break;
default:
    System.out.println("¡Un nuevo profe!");
break;
}
```

Switch desde Java 12:

```
switch (edad) {
case 1 -> System.out.println("¡Tienes 1 año!");
case 20 -> System.out.println("Tienes 20 años!");
default -> System.out.println("Tu edad no es 1 ni 20");
}
```

Recuerda que esta nueva sintaxis está deshabilitada por defecto, debemos hacer algunas configuraciones en nuestro IDE para que podamos utilizarla.

Funciones

¿Para qué sirven las funciones?

Las funciones nos ayudan a ejecutar código que dependiendo de las opciones que le enviemos, transformará y devolverá un cierto resultado. Gracias a las funciones podemos organizar, modularizar, reutilizar y evitar repetidos en nuestro código.

Todas nuestras funciones deben tener un nombre. Opcionalmente, pueden recibir argumentos y devolver un resultado. También debemos especificar el tipo de dato de nuestros argumentos y el resultado final de nuestra función.

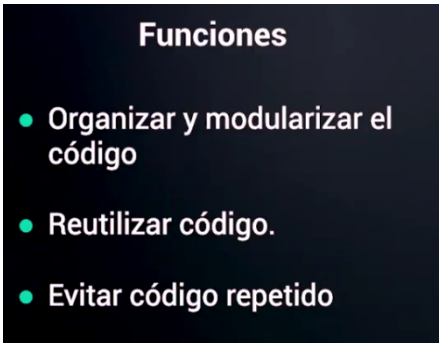
Por ejemplo:

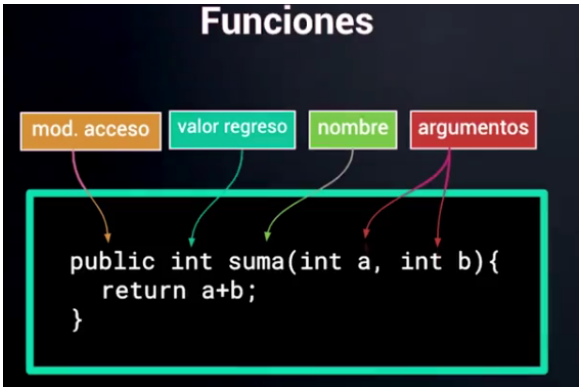
```
public int suma(inta,int b) {
return a + b;
}
```

Si nuestra función NO devuelve ningún tipo de dato podemos usar la palabra reservada `void`.

Para utilizar nuestras funciones solo debemos asignar el resultado de la función y sus parámetros a una variable con el mismo tipo de dato de la función:

```
int c = suma(5, 7);
```





Java Docs

Los Java Docs son una herramienta usada por muchas otras herramientas y aplicaciones porque nos ayuda a documentar todo nuestro código usando comentarios. Además, nos permite visualizar la documentación en formato HTML.

```
// Comentarios de una sola línea

/* Comentario
* en múltiples
* líneas */

/**
* Comentario para Java Docs
* */
```

Tags Java Docs

Syntax	Description	Description
@author	Es el autor de la clase	@author name-text
{@code}	Muestra texto en formato de código sin que sea interpretado como código HTML	{@code text}
{@docRoot}	Indica la ruta relativa al directorio raíz del documento generado desde cualquier página generada.	{@docRoot}
@deprecated	Se pone indicando que esta API no debe usarse más	@deprecated deprecatedtext
@exception	Se indica cuando es vulnerable a lanzar una excepción, en seguida se ponen las clases de las excepciones posibles	@exception class-name description
{@inheritDoc}	Indica la herencia o implementación procedentora	
{@link}	Hace un enlace a la miembro indicado	{@link package.class#member label}
{@linkplain}	Lo mismo que en anterior excepto que la etiqueta del enlace se muestra en texto plano	{@linkplain package.class#member label}
@param	Añade parámetros con nombres específicos, seguido de su descripción.	@param parameter-name description
@return	Asigna un parámetro de retorno, seguido de su descripción	@return description
@see	Se añade para manejar referencias, o información relacionada	@see reference
@serial	Se utiliza para indicar campos o clases serializables.	@serial field-description include exclude
@serialData	Se utiliza para documentar métodos que generan una serialización	@serialData data-description
@serialField	Se utiliza para documentar objetos serializados	@serialField field-name field-type field-description
@since	Se añade en el encabezado para especificar desde cuándo fue creado	@since release
@throws	Es sinónimo con la etiqueta @exception	@throws class-name description
{@value}	Cuando es usado sin argumento se usa para especificar un campo estático en otro caso se usa para mostrar el valor constante.	{@value package.class#field}
@version	Se añade en el subtítulo con la versión especificada.	@version version-text

Bucle do While

Los bucles (ciclos) nos ayudan a ejecutar una parte de nuestro código una cantidad de veces hasta que se cumpla alguna condición y podamos continuar con la ejecución de nuestro código.

Existen diferentes bucles. Por ejemplo, el bucle `do while`:

```
do {
  // instrucciones
}while (condición);
```

Los ciclos evaluarán si la condición se cumple y cuando deje de hacerlo no ejecutarán más el código del ciclo. Las instrucciones son las encargadas de que esta condición cambie de verdadero a falso. De otra forma, si las instrucciones nunca cambian la condición, el ciclo no se detendrá nunca, lo que conocemos como un ciclo infinito.

La clase Scanner le permite a los usuarios contestar algunas preguntas para que nuestro programa actúe de una forma u otra. Para usarla solo debemos importar la clase `Scanner` de las APIs de desarrollo de Java:

```
import java.util.Scanner;

int response = 0;

Scanner sc =new Scanner(System.in);
response = Integer.valueOf(sc.nextLine());
```

Operador Ternario y Bucle While

Vamos a crear el algoritmo con la lógica necesaria para encender una lampara, emitir un mensaje y detener las luces en algún momento.

El Bucle While nos ayuda a ejecutar una parte del código mientras una condición se cumpla. Recuerda tener mucho cuidado y asegurarte de que la condición del ciclo while cambie en algún momento, de otra forma, el ciclo no se detendrá nunca y sobrecargarás tu programa:

```
while (isTurnOnLight) {
  printSOS();
}
```

Los operadores ternarios son otra forma de evaluar condiciones, así como los condicionales `IF` y ELSE`:

```
// Operador Ternario:
isTurnOnLight = (isTurnOnLight) ?false :true;

// IF y ELSE:
if (isTurnOnLight) {
  isTurnOnLight =false;
}else {
  isTurnOnLight =true;
}
```

Bucle For

El Ciclo For también nos ayuda a ejecutar una parte de nuestro código las veces que sean necesarias para que se cumpla una condición. De hecho, el ciclo `FOR` nos da muchas ayudas para lograr este resultado de la forma más fácil posible:

```
// Estructura:
for (inicialización; condición; incremento o decremento;) {
  // Instrucciones
}

// En este ejemplo el mensaje de printSOS se
// ejecutará 10 veces:
for (int i = 1; i <= 10; i++) {
  printSOS();
}
```

Break, Continue y Return

Break: En Java esta sentencia la verás en dos situaciones especificamente:

1. En un **Switch:** en esta situación break hace que el flujo del switch no continúe ejecutándose a la siguiente comparación, esto con el objetivo de que solo se cumpla una sola condición:

```
switch (colorModeSelected){
case "Light":
System.out.println("Seleccionaste Light Mode");
break;
case "Night": //Ambar
System.out.println("Seleccionaste Night Mode");
break;
case "Blue Dark":
System.out.println("Seleccionaste Blue Dark Mode");
```

```
break;
}
```

2. Para salir de un **bucle:** Como acabamos de ver un break es capaz de detener el flujo en el código, en este caso detendremos el ciclo como tal terminándolo y haciendo que saltemos a la siguiente instrucción después del ciclo.

Continue: en cierto modo también nos va a servir para detener un ciclo pero en lugar de terminarlo como en el caso de break, este volverá directo a la condición.

Return: Aunque en algunos lenguajes esta sentencia sirve como un tipo goto, dónde se rompe el flujo del programa la mejor forma de usarlo en Java es en **Funciones**, cuando lo usamos aquí siempre viene acompañado de un valor, el cuál indica el dato que se estará devolviendo en la función.

Arrays

Los arreglos o *arrays* son objetos en los que podemos guardar más de una variable, una lista de elementos. Los arrays son de una sola dimensión, pero si guardamos arrays dentro de otros arrays podemos obtener arrays multidimensionales.

Los arrays se definen en código de las siguientes maneras:

```
// 1. Definir el nombre de la variable y el tipo de datos
//  que va a contener, cualquiera de las siguientes dos
// opciones es válida:
TipoDato[] nombreVariable;
TipoDato nombreVariable[];

// 2. Definir el tamaño del array, la cantidad de elementos
// que podemos guardar en el array:
TipoDato[] nombreVariable =new TipoDato[capacidad];

// Array de dos dimensiones:
TipoDato[][] cities =new String[númeroFilas][númeroColumnas];
```

Ya que los arrays pueden guardar multiples elementos, la convención es escribir los nombres de las variables en plural.

Array List

Array => Tamaño definido

ArrayList => Tamaño dinámico

```
import java.util.ArrayList;
import java.util.List;

public class ArraysTest {
    public static void main(String[] args) {
        List<String> Days = new ArrayList<>();
        Days.add("Monday");
        Days.add("Tuesday");

        Days.forEach((n) -> System.out.println(n));
    }
}
```

Ciclos For anidados

Los ciclos `FOR` nos ayudan a ejecutar una parte de nuestro código todas las veces que sean necesarias hasta que una condición se cumpla, por ejemplo, que un número supere o iguale cierta cantidad.

Eso es exactamente lo que necesitamos para trabajar con índices. En vez de escribir todos los números a mano, vamos a utilizar un ciclo para imprimir el valor de cada posición de nuestros arreglos, incluso si estos son multidimensionales:

```
// Array de una sola dimensión:
for (int i = 0; i <= 3; i++) {
  System.out.println(i);
}
// El resultado será: 0, 1, 2, 3

// Array de dos dimensiones:
for (int row = 0; row < cities.length; row++) {
for (int col = 0; col < cities[row].length; col++) {
    System.out.println(cities[row][col]);
  }
}
```

El ciclo `FOREACH` también nos ayuda a recorrer los elementos de un array posición por posición, solo que no tenemos control sobre el índice, el ciclo se encarga de recorrer todo el array automáticamente:

```
for (TipoDato elemento : coleccion) {
  // Instrucciones
}
```

POO