### Java printf para dar formato a los datos de salida

Vamos a ver como utilizar printf para dar formato a los datos que se imprimen por pantalla en Java.   
Este problema se nos plantea por ejemplo cuando queremos mostrar un número de tipo float o double con un número determinado de decimales y no con los que por defecto muestra Java.

A partir de la versión Java 5 se incorporan los métodos format y printf que permiten aplicar un formato a la salida de datos por pantalla.   
Ambos realizan la misma función, tienen exactamente el mismo formato y emulan la impresión con formato printf() de C.

Veamos primero varios **ejemplos de printf en Java** y después explicaremos en detalle la sintaxis de printf.  
  
Si queremos mostrar el número 12.3698 de tipo double con dos decimales:  
  
System.out.printf("%.2f %n", 12.3698);  
  
El primer % indica que en esa posición se va a escribir un valor. El valor a escribir se encuentra a continuación de las comillas.   
.2 indica el número de decimales.   
La f indica que el número es de tipo float o double. En la tabla que aparece más adelante podeis ver todos los carateres de conversión para todos los tipos de datos.  
%n indica un salto de línea. Equivale a \n. Con printf podemos usar ambos para hacer un salto de línea.

La salida por pantalla es:  
12,37  
  
Comprobamos que printf realiza un redondeo para mostrar los decimales indicados.  
Lo más común será que tengamos el valor en una variable, en ese caso si queremos escribir el valor de n con tres decimales:   
  
double n = 1.25036;  
System.out.printf("%.3f%n", n);  
Salida:  
1,250  
  
Para mostrar el signo + en un número positivo:   
double n = 1.25036;  
System.out.printf("%+.3f %n", n);  
Salida:  
+1.250  
  
Si el número a mostrar es un entero se utiliza el caracter d:  
int x = 10;  
System.out.printf("%d %n", x);  
Salida:  
10  
  
Para mostrarlo con signo:  
int x = 10;  
System.out.printf("%+d %n", x);  
Salida:  
+10  
  
Para mostrar varias variables pondremos tantos % como valores vamos a mostrar. Las variables se escriben a continuación de las comillas separadas por comas:  
  
double n = 1.25036;   
int x = 10;  
System.out.printf("n = %.2f x = %d %n", n, x);  
Salida:  
n = 1,25 x = 10

Cuando hay varias variables podemos indicar de cual de ellas es el valor a mostrar escribiendo 1$, 2$, 3$, ... indicando que el valor a mostrar es el de la primera variable que aparece a continuación de las comillas, de la segunda, etc.  
La instrucción anterior la podemos escribir así:   
System.out.printf("n = %1$.2f x = %2$d %n", n, x);   
  
Este número es opcional, si no aparece se entenderá que el primer valor proviene de la primera variable, el segundo de la segunda, etc.  
  
Si queremos mostrar el número 123.4567 y su cuadrado ambos con dos decimales debemos escribir:

double n = 123.4567;

System.out.printf("El cuadrado de %.2f es %.2f\n", n, n\*n);

Salida:

El cuadrado de 123,46 es 15241,56  
  
printf permite mostrar valores con un ancho de campo determinado. Por ejemplo, si queremos mostrar el contenido de n en un ancho de campo de 10 caracteres escribimos:  
  
double n = 1.25036;  
System.out.printf("%+10.2f %n", n);  
Salida:   
*bbbbb*+1.25   
Donde cada *b* indica un espacio en blanco.  
El 10 indica el tamaño en caracteres que ocupará el número en pantalla. Se cuentan además de las cifras del número el punto decimal y el signo si lo lleva. En este caso el número ocupa un espacio de 5 caracteres (3 cifras, un punto y el signo) por lo tanto se añaden 5 espacios en blanco al principio para completar el tamaño de 10.  
  
Si queremos que en lugar de espacios en blancos nos muestre el número completando el ancho con ceros escribimos:   
System.out.printf("%+010.2f %n", n);   
Salida:  
+000001.25

Más ejemplos de printf:  
  
Mostrar el número 1.22 en un ancho de campo de 10 caracteres y con dos decimales.

double precio = 1.22;

System.out.printf("%10.2f", precio);

Salida:

*bbbbbb*1.22    
(el carácter b indica un espacio en blanco)

El número ocupa un espacio total de 10 caracteres incluyendo el punto y los dos decimales.

Mostrar la cadena "Total:" con un ancho de 10 caracteres y alineada a la izquierda:

System.out.printf("%-10s", "Total:");

Salida:

Total:*bbbb*  
  
El caracter s indica que se va a mostrar una cadena de caracteres.  
El signo - indica alineación a la izquierda.

Mostrar la cadena "Total:" con un ancho de 10 caracteres y alineada a la derecha:

System.out.printf("%10s", "Total:");

Salida:

*bbbb*Total:  
  
Al final puedes ver un ejemplo completo con distintos usos de printf.

Veamos ahora detenidamente la sintaxis de printf:  
  
La sintaxis general de printf es:

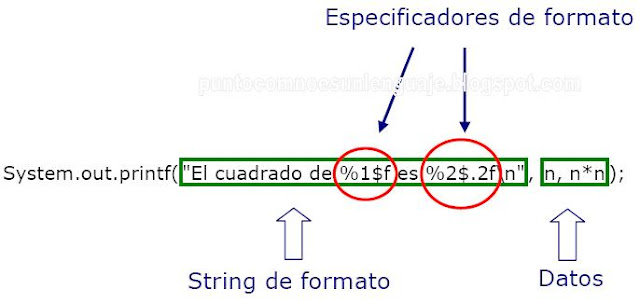
printf (String de formato, Object … datos);

El *String de formato* es una cadena de caracteres que contiene:

- texto fijo que será mostrado tal cual

- especificadores de formato que determinan la forma en que se van mostrar los datos.

*datos* representa la información que se va a mostrar y sobre la que se aplica el formato. El número de datos que se pueden mostrar es variable.

[](http://4.bp.blogspot.com/-4w2zzkUw6Uc/UBkiBIHrqKI/AAAAAAAAADc/MUETdqjCL4M/s1600/java-printf.jpg)

Explicación de cada una de las partes que aparecen en la instrucción printf:  
  
Especificadores de formato:  
  
La sintaxis para los especificadores de formato de printf es:

%[posición\_dato$][indicador\_de\_formato][ancho][.precision]carácter\_de\_conversión

Los elementos entre corchetes son opcionales.

posición\_dato: indica la posición del dato sobre el que se va aplicar el formato. El primero por la izquierda ocupa la posición 1.

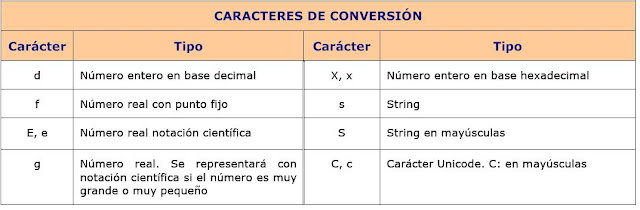
indicador\_de\_formato: es el conjunto de caracteres que determina el formato de salida. Los indicadores de formato de printf en Java son:

[](http://2.bp.blogspot.com/-I4JMiR5sMkk/UBkkvIo3ChI/AAAAAAAAADs/d5Sea-DT0JQ/s1600/java-printf-formato-indicadores.jpg)

ancho: Indica el tamaño mínimo, medido en número de caracteres, que debe ocupar el dato en pantalla.

.precision: Indica el número de decimales que serán representados. Solo aplicable a datos de tipo float o double.

carácter\_de\_conversión: Carácter que indica cómo tiene que ser formateado el dato. Los más utilizados se muestran en la tabla.

[](http://4.bp.blogspot.com/-q8INXLuKqss/UBklRnA3kLI/AAAAAAAAAD0/yIm56kaN4Bg/s1600/java-printf-formato-conversion.jpg)

Ejemplo completo con distintos usos de printf en Java:

# public static void main(String[] args) {      double q = 1.0/3.0;      System.out.printf ("1.0/3.0 = %5.3f %n", q);      System.out.printf ("1.0/3.0 = %7.5f %n", q);      q = 1.0/2.0;      System.out.printf ("1.0/2.0 = %09.3f %n", q);      q = 1000.0/3.0;      System.out.printf ("1000/3.0 = %7.1e h%n", q);      q = 3.0/4567.0;      System.out.printf ("3.0/4567.0 = %7.3e %n", q);      q = -1.0/0.0;      System.out.printf ("-1.0/0.0 = %7.2e %n", q);      q = 0.0/0.0;      System.out.printf ("0.0/0.0 = %5.2e %n", q);      System.out.printf ("pi = %5.3f, e = %10.4f %n", Math.PI, Math.E);      double r = 1.1;      System.out.printf             ("C = 2 \* %1$5.5f \* %2$4.1f, "+"A = %2$4.1f \* %2$4.1f \* %1$5.5f %n",Math.PI, r); }  Salida: 1.0/3.0 = 0,333 1.0/3.0 = 0,33333 1.0/2.0 = 00000,500 1000/3.0 = 3,3e+02 h 3.0/4567.0 = 6,569e-04 -1.0/0.0 = -Infinity 0.0/0.0 = NaN pi = 3,142, e = 2,7183 C = 2 \* 3,14159 \* 1,1, A = 1,1 \* 1,1 \* 3,14159

# Javac Miclase.java

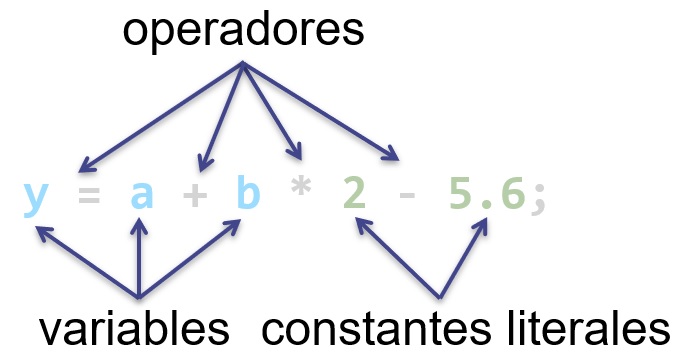
# java -cp . Miclase

# Expresiones y sentencias

Las **expresiones** permiten combinar **datos** y **operadores** para calcular otros datos, es decir, obtener un resultado.

Además, pueden contener **palabras clave** del lenguaje y **signos de puntuación**.

Un **operador** es un **símbolo** que determina la operación a realizar sobre los **operandos** (datos) a los que afecta.

[](https://www2.eii.uva.es/fund_inf/cpp/_images/operadores_variables.jpg)

Una **sentencia** es un conjunto de expresiones que permiten **ejecutar** una determinada acción.

En java las **sentencias simples** se caracterizan por terminar con el signo de puntuación ;.

(x + 3)\*2 // expresión

a > b && c < d // expresión

int x; // sentencia

y = (x + 3)\*2; // sentencia

Expresiones, sentencias y bloques

Ahora que ya entiende las variables y los operadores es el momento de aprender lo que son las *expresiones*, las *sentencias* y los *bloques*. Los operadores se utilizan para construir expresiones que calculan valores; las expresiones son los componentes centrales de las sentencias; las sentencias se pueden agrupar en bloques.

### Expresiones

Una expresión es una construcción hecha de variables, operadores e invocaciones de métodos, creados según la sintaxis del lenguaje, que evalúa a un solo valor. Ya hemos visto ejemplos de expresiones, ilustradas en negrita a continuación:

int **cadence = 0**;

**anArray[0] = 100**;

System.out.println(**"Elemento 1 en el índice 0:** **" + anArray[0]**);

int **result = 1 + 2**; // result es ahora 3

if(**value1 == value2**) System.out.println(**"value1 == value2"**);

El tipo de dato del valor devuelto por una expresión depende de los elementos utilizadas en ella. La expresión cadence = 0 devuelve un int porque el operador de asignación devuelve un valor del mismo tipo de dato que el operando de su izquierda; en este caso cadence es un int. Como puede observar en los demás ejemplos, una expresión puede devolver también otro tipo de datos, por ejemplo boolean o String.

El lenguaje de programación Java permite construir expresiones compuestas a partir de varias expresiones más pequeñas siempre que el tipo de dato requerido por una parte de la expresión concuerde con el de la otra. A continuación un ejemplo de una expresión compuesta:

1 \* 2 \* 3

En este ejemplo el orden en el que se evalúa la expresión no es importante porque el resultado de la muliplicación no depende del orden; el resultado es siempre el mismo sin importar el orden en que se apliquen las multiplicaciones. Sin embargo esto no es cierto para todas las expresiones. Por ejemplo, la siguiente expresión dará resultados diferentes según se realice primero la operación de suma o la de división:

x + y / 100 // ambíguo

Puede indicar exáctamente cómo se debe evaluar una expresión mediante el uso de paréntesis: ( y ). Para eliminar la ambigüedad de la expresión anterior se podría escribir de la siguiente forma:

(x + y) / 100 // no ambíguo, recomendado

Si no indica explícitamente el orden de las operaciones que se deben realizar, el orden lo determina la precedencia asignada a los operadores utilizados en la expresión. Lo peradores con mayor precedencia se evalúan primero. Por ejemplo, el operador de división tiene mayor precedencia que el de suma. Por lo tanto las siguientes dos sentencias son equivalentes:

x + y / 100   
x + (y / 100) // no ambíguo, recomendado

Sea explícito al escribir expresiones compuestas e indique con paréntesis qué operadores se deben evaluar primero. Esta práctica hará que su código sea más fácil de leer y mantener.

### Sentencias

Las sentencias son, a grandes rasgos, equivalentes a las oraciones del lenguaje natural. Una sentencia forma una unidad de ejecución completa. Los siguientes tipos de expresiones se pueden convertir en sentencia si se terminan con punto y coma (;).

* Expresiones de asignación
* Cualquier uso de code>++ o --
* Invocaciones de métodos
* Expresiones de creación de objetos

Este tipo de sentencias se llaman sentencias de expresión. Veamos algunos ejemplos de sentencias de expresión.

unValor = 8933.234; // sentencia de asignación

unValor++; // sentencia de incremento

System.out.println("¡Hola Mundo!"); // sentencia de invicación de método

Bicycle miBici = new Bicycle(); // sentencia de creación de objeto

Hay dos otros tipo de sentencias además de las de expresión: *sentencias de declaración* y *sentencias de control de flujo*. Una sentencia de declaración declara una variable. Ya hemos visto muchos ejemplos de sentencias de declaración:

double unValor = 8933.234; //sentencia de declaración

Finalmente, las sentencias de control de flujo regulan el orden en el que se ejecutan las sentencias. Aprenderá más acerca de las sentencias de control de flujo en la siguiente sección, [Sentencias de control de flujo](https://java.codexion.com/java/nutsandbolts/flow.html)

### Bloques

Un *bloque* es un grupo de sentencias contenido entre corchetes emparejados que se puede utilizar en cualquier lugar en el que se permita una sentencia sencilla. El siguiente ejemplo, [BlockDemo](https://java.codexion.com/java/nutsandbolts/examples/BlockDemo.java), ilustra el uso de bloques:

class BlockDemo {

public static void main(String[] args) {

boolean condition = true;

if (condition) { **// empieza el bloque 1**

System.out.println("Condition es verdadero.");

} **// fin del bloque one**

else { **// empieza el bloque 2**

System.out.println("Condition es falso.");

} **// fin del bloque 2**

}

}

Preguntas y ejercicios: Expresiones, sentencias y bloques

**Preguntas**

1. Los operadores se pueden usar en la construcción de \_\_\_, que calculan valores.
2. Las expresiones son los componentes centrales de \_\_\_.
3. Las sentencias se pueden agrupar en \_\_\_.
4. El siguiente fragmento de código es un ejemplo de una expresión \_\_\_.
5. 1 \* 2 \* 3
6. Las sentencias son, a grandes rasgos, equivalentes a las oraciones del lenguaje natural, pero en vez de terminar con un punto una sentencia termina con \_\_\_.
7. Un bloque es un grupo de sentencias contenido entre \_\_\_ emparejados que se puede utilizar en cualquier lugar en el que se permita una sentencia sencilla.

**Ejercicios**

Identifique los siguientes tipos de expresiones:

* unValor = 8933.234;
* unValor++;
* System.out.println("¡Hola Mundo!");
* Bicycle miBici = new Bicycle();

**Caracteres y secuencias de escape de Java**

En este tema vamos a estudiar los caracteres que pueden aparecer en un programa Java. Ya sabemos como crear un programa básico, ahora veremos los caracteres que podemos usar para construir programas más complejos.

**Character set: el conjunto de caracteres de Java**

Para escribir un programa Java como en cualquier otro lenguaje de programación utilizamos los caracteres. Como es lógico, hay una serie de restricciones y no podemos utilizar cualquier carácter que queramos a la hora de escribir un programa. Cada lenguaje determina el conjunto de caracteres (character set) que se pueden utilizar para escribir el código. En Java el conjunto de caracteres no se limita a los que contiene el código ASCII como pasa en algunos lenguajes de programación. Java soporta el sistema de codificación UNICODE y esto supone que el conjunto de caracteres del lenguaje sea muy amplio. ASCII es un subconjunto de UNICODE por lo tanto los caracteres ASCII siguen siendo válidos en Java.  
Los caracteres que pueden aparecer en un programa Java para formar las constantes, variables, expresiones, etc., son:

* Las **letras** **mayúsculas y minúsculas** de la A(a) a la Z(z) de los alfabetos internacionales. Los caracteres ñ y Ñ son válidos.
* **Dígitos:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
* Los caracteres  ' \_ ' '$'  y cualquier otro carácter considerado como letra en el sistema de codificación Unicode.
* Los **caracteres especiales** y signos de puntuación siguientes:

+ - \* / = % & # ! ? ^ “ ‘ ~ \ | < > ( ) [ ] { } : ; . ,

* **Separadores**:
  + espacio
  + tabulador
  + salto de línea
* **Secuencias de escape:** por ejemplo \t,  \n. Se explican a continuación

Los separadores como el tabulador y el salto de línea ayudan a que el programa sea más legible por las personas.  
Por ejemplo, podemos escribir el método main sin utilizar espacios en blanco, de la forma:

public static void main(String [] args){System.out.println("Hola Mundo!!!");}

Aunque queda mucho más claro si introducimos tabuladores y saltos de línea:

public static void main(String [] args){

System.out.println("Hola Mundo!!!");

}

**Secuencias de escape**

Una secuencia de escape esta formada por una barra inversa seguida de una letra, un carácter o de una combinación de dígitos.

Una secuencia de escape siempre representa un solo carácter aunque se escriba con dos o más caracteres.   
Se utilizan para realizar acciones como salto de línea o para usar caracteres no imprimibles.   
Algunas secuencias de escape definidas en Java son:

|  |  |
| --- | --- |
| **Secuencia de escape** | **Descripción** |
| \n | Salto de línea. Sitúa el cursor al principio de la línea siguiente |
| \b | Retroceso. Mueve el cursor un carácter atrás en la línea actual. |
| \t | Tabulador horizontal. Mueve el cursor hacia adelante una distancia determinada por el tabulador. |
| \r | Ir al principio de la línea. Mueve el cursor al principio de la línea actual. |
| \f | Nueva página. Mueve el cursor al principio de la siguiente página. |
| \" | Comillas. Permite mostrar por pantalla el caracter *comillas dobles*. |
| \' | Comilla simple. Permite mostrar por pantalla el carácter *comilla simple*. |
| \\ | Barra inversa. |
| \udddd | Carácter Unicode. d representa un dígito hexadecimal del caracter Unicode. |
| \s | Caracter espacio en blanco |

**Ejemplos de uso de las secuencias de escape en Java**

A continuación veremos algunos ejemplos de uso de las secuencias de escape para poder entender mejor para qué sirven:  
  
Ejemplo 1: uso de la secuencia de escape \n o salto de línea. Provoca un salto al principio de la línea siguiente en el lugar donde se coloca.

System.out.println("Juan\nVictor\nAlfonso\nEnrique");

Salida por pantalla:

Juan  
Victor  
Alfonso  
Enrique

Ejemplo 2: Uso de la secuencia de escape \r. Provoca que el cursor se sitúe al principio de la línea siguiente.

System.out.println("Lunes\rMartes, Miércoles");

Salida por pantalla:

Martes, Miércoles

Ejemplo 3: Uso de la secuencia de escape \b. Provoca que el cursor retroceda un carácter.

System.out.println("Lunes\bMartes");

Salida por pantalla:

LuneMartes

Ejemplo 4: Uso de la secuencia de escape \t. Provoca que el cursor avance una distancia determinada.

System.out.println("Lunes\tMartes\tMiércoles");

Salida por pantalla:

Lunes    Martes    Miércoles

Ejemplo 5: Uso de las secuencias de escape \" y \'. Permite mostrar estos caracteres por pantalla.

System.out.println("\"Lunes\",\"Martes\",\'Miércoles\'");

Salida por pantalla:

"Lunes","Martes",'Miércoles'

## SECUENCIAS DE ESCAPE PARA LOS COLORES DE LA PANTALLA

Cuando programamos en **Java** estamos acostumbrados a **enviar mensajes por la consola** para informar a los usuarios de lo que va a ocurriendo. Para ello la manera por defecto de conseguirlo es mediante el **uso del método System.out.println**.

Este método, en principio, no permite cambiar el color de lo que se emite por pantalla. Todo se ve con el color predeterminado que tengamos, generalmente blanco (o gris claro) sobre fondo negro, algo así:

https://www.campusmvp.es/recursos/image.axd?picture=/2019/1T/java-consola-color-x-defecto.png

Sin embargo, cuando estamos creando una herramienta de línea de comandos nos puede resultar muy útil **emplear otros colores para destacar cosas**. Por ejemplo, usar el color rojo cuando se produzca una excepción y queramos avisar de ello tras haberla controlado, o el verde para cuando se lleve a cabo una tarea con éxito.

**Nota**: este tipo de usos del color para marcar significados **no es accesible** y por tanto no son apropiados si queremos que los puedan usar usuarios con problemas de visión, como por ejemplo los daltónicos que no distinguen entre rojo y verde. Asegúrate de que además de los colores **utilizas algo más en el propio texto** (mayúsculas, símbolos...) para indicar que se trata de un evento especial.

La buena noticia es que existen una serie de [códigos de escape ANSI](https://es.wikipedia.org/wiki/C%C3%B3digo_escape_ANSI) que nos permiten controlar ciertos aspectos de la consola, entre ellos los colores utilizados.

Si definimos las siguientes constantes en nuestro código:

public static final String ANSI\_BLACK = "\u001B[30m";  
public static final String ANSI\_RED = "\u001B[31m";  
public static final String ANSI\_GREEN = "\u001B[32m";  
public static final String ANSI\_YELLOW = "\u001B[33m";  
public static final String ANSI\_BLUE = "\u001B[34m";  
public static final String ANSI\_PURPLE = "\u001B[35m";  
public static final String ANSI\_CYAN = "\u001B[36m";  
public static final String ANSI\_WHITE = "\u001B[37m";

Podremos utilizarlas para controlar el color que queremos mostrar con System.out.println.

Además podemos definir esta constante:

public static final String ANSI\_RESET = "\u001B[0m";

que nos permitirá dejarlo todo como estaba y volver a los colores por defecto.

Así, podemos escribir lo siguiente:

System.out.println(ANSI\_RED + "Texto de color rojo" + ANSI\_RESET);

y veremos el color rojo como esperábamos:

https://www.campusmvp.es/recursos/image.axd?picture=/2019/1T/java-consola-color-rojo.png

dejando además la consola en su estado predeterminado, justo a continuación.

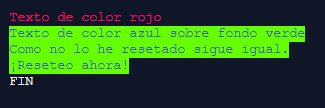
Del mismo modo podemos definir una serie de constantes para establecer el color de fondo que queremos usar con el texto, así:

public static final String ANSI\_BLACK\_BACKGROUND = "\u001B[40m";public static final String ANSI\_RED\_BACKGROUND = "\u001B[41m";public static final String ANSI\_GREEN\_BACKGROUND = "\u001B[42m";public static final String ANSI\_YELLOW\_BACKGROUND = "\u001B[43m";public static final String ANSI\_BLUE\_BACKGROUND = "\u001B[44m";public static final String ANSI\_PURPLE\_BACKGROUND = "\u001B[45m";public static final String ANSI\_CYAN\_BACKGROUND = "\u001B[46m";public static final String ANSI\_WHITE\_BACKGROUND = "\u001B[47m"

Con ellas podemos establecer también el color de fondo y combinarlas con las anteriores para el color del texto, así:

System.out.println(ANSI\_GREEN\_BACKGROUND + ANSI\_BLUE + "Texto de color azul sobre fondo verde");System.out.println("Como no lo he resetado sigue igual.");System.out.println("¡Reseteo ahora!" + ANSI\_RESET);System.out.println("FIN");

obteniendo esto por pantalla:



Si te fijas, mientras no le mandes el código de control de "reset" no se vuelve a establecer el color por defecto, por lo que podrías ir enviando comandos no visibles para cambio de color sin necesidad de escribir nada, y usarlos simplemente para ir controlando el cambio de colores.

**¿Qué son las Enumeraciones en Java?**

En Java, las enumeración se denominan **enum**. Un objeto de un tipo de enumeración sólo puede llegar a contener los valores definidos por la lista. Debido a esto, una enumeración **brinda una manera de definir con precisión un nuevo tipo de datos** que tiene un número fijo de valores que se consideran válidos.

Para comprender mejor esto, supongamos que tenemos una baraja inglesa. La misma se compone de 4 palos, los cuales serán 4 enumeradores llamados Flores, Diamantes, Corazones y Espadas, que pertenecen a un tipo enumerado llamado Cartas.

Las enumeraciones son muy **útiles siempre que se necesite definir un conjunto de valores que representen una colección de elementos**. En el pasado, dichos valores se definían como variables finales, pero actualmente las enumeraciones ofrecen un enfoque más estructurado.

**Declaración de enum en Java**

La **declaración de Enum** puede hacerse **dentro o fuera de una clase**, **pero nunca dentro de un método**. Las enumeraciones se crean usando la palabra clave **enum**.

Al efectuar una declaración de enum, la primera línea de código debe ser una lista de constantes y luego otras cosas como métodos, variables y constructores.

Según las convenciones de nomenclatura de Java, se recomienda que l**os nombres se escriban con las constantes con mayúsculas**.

**Ejemplo de enumerado en Java**

A continuación se pueden observar ejemplos de cómo se realiza la declaración de un enum:

//Declaración de una enumeración de los días de la semana

//Declaración de una enumeración de niveles

enum Nivel {ALTO, MEDIO, BAJO};

//Declaración de una enumeración de direcciones

enum Direccion {NORTE, SUR, ESTE, OESTE};

// Declaración de una enumeración de colores

enum Color { ROJO, VERDE, AMARILLO, AZUL};

//Una enumeración de transporte

enum Transporte{ COCHE, CAMIÓN, AVION, TREN, BARCO};

**¿Cómo implementar los Enum en Java?**

Los enum se pueden emplear:

* en estructuras [condicionales de tipo **if**](https://open-bootcamp.com/cursos/java/sentencias-if-else).
* con [**Switch**](https://open-bootcamp.com/cursos/java/switch), un condicional de selección de Java.
* en llamadas a los Métodos Asignación a variables Iteradores.
* con el método [**toString()**](https://open-bootcamp.com/cursos/java/metodos-de-la-clase-string), **valueOf()**.

**Programa con enum en Java**

En este simple programa se puede apreciar la sintaxis del código para emplear un enum en Java usando una estructura condicional de tipo **if**.

Primero se debe hacer la declaración del enum como se vio en los ejemplos anteriores, enumerando los **identificadores** *PUBLICIDAD*, *BORRADOR* y *CANCELADO* que se denominan **constantes de enumeración.**

Luego se puede declarar un [método **main**](https://open-bootcamp.com/cursos/java/metodo-main-y-systemout)**()** dentro una clase **class()** donde cada constante se declara implícitamente como un miembro público (**public**) y estático (**static**) de *Status*. Dentro del método main() las constantes de enumeración se usan en una estructura condicional de tipo **if**, donde se ejecutarán ciertas líneas de código, según se valide la igual expresada como condición en la sintaxis.

//Declaración del enumeración de status

enum Status {PUBLICADO, BORRADOR,CANCELADO}

Class Main {

Public static void main (String args[]){

Status estado = Status.PUBLICADO;

if (estado = Status. CANCELADO){

//Ya declarado el enum, aquí dentro se puede escribir el código del programa

que se necesite.

}

}

}

# Declaración y uso de constantes en Java. Uso del final en Java, datos final

En la sección anterior hablamos sobre [variables y tipos en java](https://www.programarya.com/Cursos/Java/Sistema-de-Tipos/variables-y-tipos) y aprendimos cómo se declara una variable, cómo se le asigna un tipo de dato específico (primitivo o complejo). En esta sección hablaremos ya no tanto de "variables" sino más bien de constantes. Una constante desde el punto de vista de la programación es un dato cuyo valor no cambia durante la ejecución del programa, en otras palabras, una vez que a una constante se le asigna un valor, este no podrá ser modificado y permanecerá así durante toda la ejecución del programa.

Las constantes son útiles para datos o atributos para los cuales el valor no tiene por qué cambiar, por ejemplo, el número del documento de identidad de una persona, lo más normal es que dicho número no cambie o la fecha de nacimiento, una persona tiene una fecha de nacimiento única e invariable, por lo tanto por ejemplo un atributo "nacimiento" debería ser constante. Con esto podremos evitar modificaciones en nuestro sistema que puedan causar problemas durante la ejecución del mismo.

En esta sección no hablaremos únicamente sobre cómo hacer que un dato cualquiera no permanezca constante, sino también, veremos qué implicaciones tiene que una clase, un método y un atributo sean constantes (final). Comencemos:

## Constantes en Java:

La palabra clave que java ha reservado para definir constantes es la palabra "final". En java es muy simple definir constantes, solo basta con ponerles el modificador final antes de la declaración del tipo. Al definir un dato como constante le podremos asignar un valor por primera vez y luego de eso no será posible cambiarle ese valor. Una vez inicializado el dato, este no podrá cambiar su valor de ninguna forma.

## Modificador final en Java:

Tal como mencioné hace un momento el modificador final es la palabra reservada que Java definió para crear constantes. Veamos entonces la sintaxis general para definir un dato como constante en Java:

### Datos constantes en Java

final tipo nombreVble = valor;

Como se puede apreciar la sintaxis es bastante sencilla, sin embargo hay que tener en cuenta algunas cosas: cuando vamos a declarar un dato cualquiera como final, debemos asignarle inmediatamente un valor, es decir debemos inicializarlo de inmediato, de lo contrario no podremos compilar nuestro código. En este punto debo hacer una aclaración, un dato cualquiera que se encuentre por ejemplo al interior del método main, debe ser inicializado de inmediato, sin embargo un atributo propio de una clase no necesariamente debe ser inicializado en el mismo momento que se declara dicho atributo, sino que puede ser inicializado también por un constructor, dando así la posibilidad de que cada objeto especifico tenga su propio valor constante, ingresa a continuacón para saber más sobre [atributos y constructores](https://www.programarya.com/Cursos/Java/Objetos-y-Clases). Vemos el ejemplo:

### Atributos constantes en java

public class Constantes {

private final String atribConstante; //Notar que no ha sido inicializado

//Constructor

Constantes(String valor) {

//Aquí se inicializó el atributo atribConstante = valor; }

}

En el código anterior, podemos apreciar entonces que en el caso de los atributos de una clase no es necesario inicializar dicho atributo una vez que éste es declarado. Sin embargo no todo es tan bueno, pues si no lo hacemos al momento de declarar el atributo, entonces estaremos obligados a hacerlo en el constructor de la clase. Java debe asegurarse que el valor de una variable final sea asignado y las dos únicas posibilidades son el constructor o en la declaración misma del atributo. Existe algo particular con este comportamiento de Java y es que necesitamos que todos los constructores le den valor al atributo constante (si no lo hicimos al momento de declararlo), de este modo, si tenemos más de un constructor es obligatorio que cada uno de ellos le asigne valor al atributo constante (final), pues no basta que uno de ellos lo haga, Java nos permitirá compilar el código siempre y cuando todos y cada uno de los constructores le de valor al atributo final (constante), pues si usáramos un constructor que no le diera valor a dicho atributo, éste quedaría con valor nulo y nunca podrá ser cambiado lo cual es inconsistente. Veamos en código de lo que estoy hablando:

### Constructores y atributos constantes en Java

public class Constantes {

private int atributo;

//Este atributo no es constant

private final String atribConstante;

//Constante y no ha sido inicializado

//Primer constructor

//Ver que en éste le asigna valor al atributo

final Constantes(String valor) {

//Aquí se inicializó el atributo

atribConstante = valor;

}

//Segundo constructor

//Éste no le asigna valor al atributo

final Constantes(int valor) {

//Aquí se inicializó el atributo

atributo = valor;

//Pero no se le dio valor al atributo

final //atribConstante ="";

}

//Lo anterior es un error!!

//Se debe descomentar la linea 22 para solucionarlo }

## Métodos y clases final

El modificador final también puede ser usado en las clases y en los métodos, de manera similar a como se hace con los datos y los atributos, sin embargo el significado que esto posee es diferente.

### Final class. Clases final

Una clase final es una clase que no puede ser extendida, es decir ninguna otra clase puede ser hija de ésta, será entonces una clase única y que no tendrá herencia o herederos al no poder ser extendida. Veamos:

Declaramos una clase "Madre" que será final, esto quiere decir que no podrá ser extendida por otra

//Se declara la clase como final

public final class ClaseMadre {

//Atributos de la clase

private int atributo1;

private String atributo2;

//Método de la clase

public void metodo1() {

System.out.print("Hola");

}

}

Declaramos una clase "Hija" que intentará extender de la clase "Madre", la cual es final. Esto generará un error de compilación y no podremos ejecutar nuestro código.

//Se declara la clase como hija de madre

//Esto genera un error

public class ClaseHija extends ClaseMadre {

//Atributos de la clase

private int atributo1;

private String atributo2;

//Método de la clase

public void metodo1() {

System.out.print("Hola");

}

}

### Métodos final

Un método final no puede ser redefinido por una clase hija. Esto quiere decir que si yo defino un método final en la clase X y luego la clase Y es hija de la clase X, entonces la clase Y no podrá cambiar la definición de dicho método, puesto que éste es final. Veamos:

Declaramos una clase "Madre" que será tendrá un método final, esto quiere decir que dicho método no podrá ser redefinido.

//Se declara la clase como final public class ClaseMadre { //Atributos de la clase private int atributo1; private String atributo2; //Método final de la clase public final void metodo1() { System.out.print("Hola"); } }

Declaramos una clase "Hija" que extenderá de la clase "Madre" sin problemas e intentará redefinir el método1. Esto generará un error de compilación y no podremos ejecutar nuestro código.

//Se declara la clase como hija de madre public class ClaseHija extends ClaseMadre { //Atributos de la clase private int atributo1; private String atributo2; //Método de la clase public void metodo1() { System.out.print("Adios"); } //Lo anterior genera error, pues no se puede redefinir un método final public int metodo1(int valor) { return valor; } //Esto NO es un error, pues NO estamos redefiniendo el método //Redefinir no es lo mismo que sobrecargar. }

En el ejemplo anterior acabamos de ver como hemos intentado redefinir el método de la clase madre y esto no generaría un error, pues método1 es final. Quiero hacer la aclaración aquí. Redefinir no es lo mismo que sobrecargar. Cuando sobrecargamos un método estamos cambiando la firma del método y esto lo podemos hacer independientemente de si el método es final o no. Sin embargo redefinir el método es intentar hacer que la clase Hija tenga el mismo método (sin sobrecargarlo), pero que haga algo diferente, esto es lo que no podremos hacer. En este caso intentamos que el método mostrara en pantalla "Adios" pero no podemos hacerlo, mientras que luego SOBRECARGAMOS el método1, poniéndole un parámetro de entrada y un valor de retorno.

Sé que esto último es bastante confuso y avanzado, sin embargo quise tocar el tema para ir abriéndome a otras cosas y mostrando el alto potencial que tiene Java. En otras secciones hablaré con detalle acerca de herencia en Java, todo esto de clases Hijas, clases Madre y varias cosas más.

# Métodos en Java, funciones y procedimientos. Cómo hacerlos y usarlos

Los métodos en Java, las funciones y los procedimientos, especialmente en Java, son una herramienta indispensable para programar. Java nos permite crear o hacer nuestros propios métodos y usarlos sencillamente como también nos facilita hacer uso de los métodos de otras [librerías](https://www.programarya.com/Cursos/Java/Librerias) (funciones matemáticas, aritméticas, de archivos, de fechas, etc. Cualquiera que sea el caso, las funciones permiten automatizar tareas que requerimos con frecuencia y que además se pueden generalizar por medio de parámetros o argumentos. Aprender a crear métodos en Java y usarlos correctamente es de gran importancia, separar nuestro código en módulos y según las tareas que requerimos. En java una función debe contener la implementación de una utilidad de nuestra aplicación, esto nos pide que por cada utilidad básica (abrir, cerrar, cargar, mover, etc.) sería adecuado tener al menos una función asociada a ésta, pues sería muy complejo usar o crear un método que haga todo de una sola vez, por esto es muy buena idea separar cada tarea en una función o método (según corresponda).

Para estar claros en todo, en Java es mucho más común hablar de métodos que de funciones y procedimientos y esto se debe a que en realidad un método, una función y un procedimiento NO son lo mismo, veamos la diferencia:

## ¿Funciones, métodos o procedimientos?

Es muy común entre programadores que se hable indistintamente de estos tres términos sin embargo poseen deferencias fundamentales.

### Funciones:

Las funciones son un conjunto de líneas de código (instrucciones), encapsulados en un bloque, usualmente reciben parámetros, cuyos valores utilizan para efectuar operaciones y adicionalmente retornan un valor. En otras palabras una función puede recibir parámetros o argumentos (algunas no reciben nada), hace uso de dichos valores recibidos como sea necesario y retorna un valor usando la instrucción return, si no retorna algo, entonces no es una función. En java las funciones usan el modificador static.

### Métodos:

Los métodos y las funciones en Java están en capacidad de realizar las mismas tareas, es decir, son funcionalmente idénticos, pero su diferencia radica en la manera en que hacemos uso de uno u otro (el contexto). Un método también puede recibir valores, efectuar operaciones con estos y retornar valores, sin embargo en método está asociado a un objeto, SIEMPRE, básicamente un método es una función que pertenece a un objeto o clase, mientras que una función existe por sí sola, sin necesidad de un objeto para ser usada. **Nota:** Es aquí donde digo que en Java se debe hablar de métodos y no de funciones, pues en Java estamos siempre obligados a crear un objeto para usar el método. Para que sea una función esta debe ser static, para que no requiera de un objeto para ser llamada.

### Procedimientos:

Los procedimientos son básicamente un conjunto de instrucciones que se ejecutan sin retornar ningún valor, hay quienes dicen que un procedimiento no recibe valores o argumentos, sin embargo en la definición no hay nada que se lo impida. En el contexto de Java un procedimiento es básicamente un método cuyo tipo de retorno es void que no nos obliga a utilizar una sentencia return.

## Crear un método en Java

La sintaxis para declarar una función es muy simple, veamos:

[acceso] [modificador] tipo nombreFuncion([tipo nombreArgumento,[tipo nombreArgumento]...]) { /\* \* Bloque de instrucciones \*/ return valor; }

EL primer componente corresponde al [modificador de acceso](https://www.programarya.com/Cursos/Java/Modificadores-de-Acceso), que puede ser public o private, éste es opcional, si no ponemos nada, se asume el modificador de acceso por defecto, el segundo componente es el modificador que puede ser [final o static](https://www.programarya.com/Cursos/Java/Sistema-de-Tipos/Final-y-Constantes) (o ambas), también es opcional. Recordemos que un método o función siempre retorna algo, por lo tanto es obligatorio declararle un tipo (el tercer componente de la sintaxis anterior), puede ser entero (int), booleano (boolean), o cualquiera que consideremos, inclusive tipos complejos, luego debemos darle un nombre a dicha función, para poder identificarla y llamarla (invocarla) durante la ejecución, después al interior de paréntesis, podemos poner los argumentos o parámetros. Luego de la definición de la "firma" del método, se define su funcionamiento entre llaves; todo lo que esté dentro de las llaves es parte del cuerpo del método y éste se ejecuta hasta llegar a una instrucción return.

### Acerca de los argumentos o parámetros

Hay algunos detalles respecto a los argumentos de un método, veamos:

* Una función, un método o un procedimiento pueden tener una cantidad cualquier de parámetros, es decir pueden tener cero, uno, tres, diez, cien o más parámetros. Aunque habitualmente no suelen tener más de 4 o 5.
* Si una función tiene más de un parámetro cada uno de ellos debe ir separado por una coma.
* Los argumentos de una función también tienen un tipo y un nombre que los identifica. El tipo del argumento puede ser cualquiera y no tiene relación con el tipo del método.
* Al recibir un argumento nada nos obliga a hacer uso de éste al interior del método, sin embargo para qué recibirlo si no lo vamos a usar.
* En Java los parámetros que podemos recibir pueden ser por valor por referencia, esto implica que si modificamos los valores recibidos al interior del método, estos pueden mantener sus cambios o no después de ejecutada el método (esto lo explico con más detalla enseguida).

### Consejos acerca de return

Debes tener en cuenta dos cosas importantes con la sentencia return:

* Cualquier instrucción que se encuentre después de la ejecución de return NO será ejecutada. Es común encontrar funciones con múltiples sentencias return al interior de condicionales, pero una vez que el código ejecuta una sentencia return lo que haya de allí hacia abajo no se ejecutará.
* El tipo del valor que se retorna en una función debe coincidir con el del tipo declarado a la función, es decir si se declara int, el valor retornado debe ser un número entero.
* En el caso de los procedimientos (void) podemos usar la sentencia return pero sin ningún tupo de valor, sólo la usaríamos como una manera de terminar la ejecución del procedimiento.

Veamos algunos ejemplos.

## Ejemplos de métodos

Veamos algunos ejemplos prácticos de métodos en Java.

### Ejemplo 1:

int metodoEntero()//Función sin parámetros { int suma = 5+5; return suma; //Acá termina la ejecución del método //return 5+5;//Este return nunca se ejecutará //Intenta intercambiar la línea 3 con la 5 //int x = 10; //Esta línea nunca se ejecutará }

Como puedes ver es un ejemplo sencillo, es un método llamado metodoEntero, si ejecutas esto, la función te retornará el valor de suma que es 10 (5+5). Las líneas posteriores no se ejecutarán nunca, aunque no generan error alguno, no tienen utilidad. Puedes notar que para este caso es lo mismo haber escrito return suma que escribir return 5+5. Ambas líneas funcionan equivalentemente.

**Nota:** Recuerda que en Java todo debe estar al interior de una clase (o interfaz) y que debes tener al menos la función main.

### Ejemplo 2:

public String metodoString(int n)//método con un parámetro { if(n == 0)//Usamos el parámetro en la función { return "a"; //Si n es cero retorna a //Notar que de aquí para abajo no se ejecuta nada más } return "x";//Este return sólo se ejecuta cuando n NO es cero }

Aquí creamos un método público, hicimos uso de múltiples sentencia return y aprovechamos la característica de que al ser ejecutadas finalizan inmediatamente la ejecución de la parte restante del método. De este modo podemos asegurar que la función retornará "a" únicamente cuando el valor del parámetro n sea cero y retornará un "x" cuando dicho valor no sea cero.

### Ejemplo 3:

static boolean metodoBoolean(boolean n, String mensaje)//Método con dos parámetros { if(n)//Usamos el parámetro en el método { System.out.println(mensaje);//Mostramos el mensaje } return n; //Usamos el parámetro como valor a retornar }

Aquí ya tenemos una función (digo función y no método porque es static) que recibe dos parámetros, uno de ellos es usado en el condicional y el otro para mostrar su valor por pantalla con System.out.println, esta vez retornamos valores booleanos true o false y utilizamos el valor propio recibido en el parámetro. Toma en cuenta que en esta ocasión únicamente usamos una sentencia return, pues usar una al interior del if habría sido innecesario y el resultado sería el mismo.

## Hablemos un poco de los procedimientos

Los procedimientos son similares a las funciones, aunque más resumidos. Debido a que los procedimientos no retornan valores, no hacen uso de la sentencia return para devolver valores y no tienen tipo específico, sólo void. Veamos un ejemplo:

### Ejemplo de procedimientos

void procedimiento(int n, String nombre) //Notar el void { if(n > 0 && !nombre.equals(""))//usamos los dos parámetros { System.out.println("hola " + nombre); return; //Si no ponemos este return se mostraría hola y luego adiós } //También podríamos usar un else en vez del return System.out.println("adios"); }

De este ejemplo podemos ver que ya no se usa un tipo sino que se pone void, indicando que no retorna valores, también podemos ver que un procedimiento también puede recibir parámetros o argumentos.

**Recuerda:** Los procedimientos también pueden usar la sentencia return, pero no con un valor. En los procedimientos el return sólo se utiliza para finalizar allí la ejecución.

## Invocando funciones y procedimientos en Java

Ya hemos visto cómo hacer funciones en Java, cómo se crean y cómo se ejecutan, ahora veamos cómo usar un método, función o procedimiento.

nombre([valor,[valor]...]);

Como puedes notar es bastante sencillo invocar o llamar funciones en Java, sólo necesitas el nombre del método, función o procedimiento y enviarle el valor de los parámetros. Hay que hacer algunas salvedades respecto a esto.

### Detalles para invocar métodos funciones y procedimientos

* No importa si se trata de un método en Java o de una función o de un método, sólo debes ocuparte de enviar los parámetros de la forma correcta para invocarlos.
* El nombre debe coincidir exactamente al momento de invocar, pues es la única forma de identificarlo.
* El orden de los parámetros y el tipo debe coincidir. Hay que ser cuidadosos al momento de enviar los parámetros, debemos hacerlo en el mismo orden en el que fueron declarados y deben ser del mismo tipo (número, texto u otros).
* Cada parámetro enviado también va separado por comas.
* Si una función no recibe parámetros, simplemente no ponemos nada al interior de los paréntesis, pero SIEMPRE debemos poner los paréntesis.
* Invocar una función sigue siendo una sentencia común y corriente en Java, así que ésta debe finalizar con ';' como siempre.
* El valor retornado por un método o función puede ser asignado a una variable del mismo tipo, pero no podemos hacer esto con un procedimiento, pues no retornan valor alguno.
* Una función puede llamar a otra dentro de sí misma o incluso puede ser enviada como parámetro a otra (mira el siguiente ejemplo).

## Ejemplos de uso de funciones

En el siguiente código vamos a hacer un llamado a algunas de las funciones y al procedimiento, que declaramos anteriormente.

public class Ejemplos { public static void main(String args[])//Siempre necesitamos un main { Ejemplos ejemplo = new Ejemplos(); //Cuando no es estático, debe usarse un método //Llamando a un método sin argumentos, usando el objeto ejemplo.metodoEntero(); //Asignando el valor retornado a una variable boolean respuesta = metodoBoolean(true, "hola"); // El procedimiento no es static, así que debe llamarse desde el objeto. ejemplo.procedimiento(0, "Juan");//Invocando el procedimiento //Usando una función como parámetro ejemplo.procedimiento(metodoBoolean(1, "hola"), "Juan"); //Lo que retorne metodoBoolean (en este caso 1) se envía al procedimiento } }

En el código anterior podemos ver cómo todo ha sido invocado al interior la función main (la función principal), esto nos demuestra que podemos hacer uso de funciones al interior de otras. También vemos cómo se asigna el valor retornado por el método a la variable 'respuesta' y finalmente, antes del return, vemos cómo hemos usado el valor retornado por 'funcionBool' como parámetro del procedimiento.

## Ejercicios resueltos de métodos en Java

Vamos a realizar un par de ejercicios sobre métodos y los vamos a resolver y explicar :D.

### Funciones anidadas

Para este ejercicio vamos a crear una función que llama a otra al interior de ella (por eso las quise llamar anidadas). Es un sistema de validación de un usuario que recibe un usuario y una contraseña y según sean válidos o no, muestra un mensaje al usuario.

public class Ejercicios { public static String saludar(String nombre) { //Se crea el mensaje de saludo String saludo = "Hola. Bienvenido " + nombre; return saludo;//Se retorna el saludo } public static String error(String nombre) { //Se crea el mensaje de error String error = "Ups. No pudimos validar tus datos. " + nombre + " es tu usuario?"; return error; //Se retorna el error } public static void verificar(String usuario, String contrasenia) { String usuarioValido = "JuanDMeGon"; String contraseniaValida = "MiPass"; //Se validan los datos if(usuarioValido.equals(usuario) && contraseniaValida.equals(contrasenia)) { //Si son validos se llama ala función saludar y se muestra el mensaje retornado por pantalla System.out.println(saludar(usuario)); return; //Terminamos la ejecución } //Si no son válidos entonces mostramos el mensaje de error de la funcion error. System.out.println(error(usuario)); } public static void main(String[] args) { String usuario = "Juan"; String contrasenia = "pass"; //Se hace la verificación verificar(usuario, contrasenia); //Mostrará el mensaje error. } }

En este ejercicio tenemos dos funciones publicas (funciones porque usamos el modificador static) y un procedimiento (porque es void) que hace la validación (muy muy básica) de unos datos de usuario y según estos, llama a una función de saludo o de error. Debes notar que el procedimiento es llamado desde la función main.

COMPONENTES MAYORES DE JAVA

**Major Components of Java**

The *Java Development Kit* (JDK) contains the minimum software you need to do Java development. Key commands include:

* javac: Converts .java source files into .class bytecode
* java: Executes the program
* jar: Packages files together
* javadoc: Generates documentation

javac -version

java -version

The javac program generates instructions in a special format called *bytecode* that the java command can run. Then java launches the *Java Virtual Machine* (JVM) before running the code. The JVM knows how to run bytecode on the actual machine it is on. You can think of the JVM as a special magic box on your machine that knows how to run your .class file within your particular operating system and hardware.

USO DEL COMPILADOR

To compile Java code with the javac command, the file must have the extension .java. The name of the file must match the name of the public class. The result is a file of bytecode with the same name but with a .class filename extension. Remember that bytecode consists of instructions that the JVM knows how to execute. Notice that we must omit the .class extension to run Zoo.class.

The rules for what a Java file contains, and in what order, are more detailed than what we have explained so far (there is more on this topic later in the chapter). To keep things simple for now, we follow this subset of the rules:

* Each file can contain only one public class.
* The filename must match the class name, including case, and have a .java extension.
* If the Java class is an entry point for the program, it must contain a valid main()

PASANDO PARAMETROS AL MAIN

public class MiClase {

public static void main(String[] args) {

System.out.println(**args[0]**);

System.out.println(**args[1]**);

}

}

The code args[0] accesses the first element of the array. That's right: array indexes begin with 0 in Java. To run it, type this:

javac Zoo.java

java Zoo **Bronx Zoo**

The output is what you might expect:

Bronx

Zoo

# Modificadores de acceso public, protected, default y private en Java. Encapsulamiento en Java

Los modificadores de acceso nos introducen al concepto de encapsulamiento. El encapsulamiento busca de alguna forma controlar el acceso a los datos que conforman un objeto o instancia, de este modo podríamos decir que una clase y por ende sus objetos que hacen uso de modificadores de acceso (especialmente privados) son objetos encapsulados.

Los modificadores de acceso permiten dar un nivel de seguridad mayor a nuestras aplicaciones restringiendo el acceso a diferentes atributos, métodos, constructores asegurándonos que el usuario deba seguir una "ruta" especificada por nosotros para acceder a la información.

Es muy posible que nuestras aplicaciones vayan a ser usadas por otros programadores o usuarios con cierto nivel de experiencia; haciendo uso de los modificadores de acceso podremos asegurarnos de que un valor no será modificado incorrectamente por parte de otro programador o usuario. Generalmente el acceso a los atributos se consigue por medio de los métodos get y set, pues es estrictamente necesario que los atributos de una clase sean privados.

**Nota:** Siempre se recomienda que los atributos de una clase sean privados y por tanto cada atributo debe tener sus propios métodos get y set para obtener y establecer respectivamente el valor del atributo.

**Nota 2:** Siempre que se use una clase de otro paquete, se debe importar usando [import](https://www.programarya.com/Cursos/Java/Librerias). Cuando dos clases se encuentran en el mismo paquete no es necesario hacer el import pero esto no significa que se pueda acceder a sus componentes directamente.

## Modificador de acceso private

El modificador private en Java es el más restrictivo de todos, básicamente cualquier elemento de una clase que sea privado puede ser accedido únicamente por la misma clase por nada más. Es decir, si por ejemplo, un atributo es privado solo puede ser accedido por lo métodos o constructores de la misma clase. Ninguna otra clase sin importar la relación que tengan podrá tener acceso a ellos.

package aap.ejemplo1;

public class Ejemplo1

{

private int atributo1;//Este atributo es privado

private int contador = 0; //Contador de registro

//Si un atributo es privado podemos crear método get y set ...

//... para éste y permitir el acceso a él desde otras instancias

public void setAtributo1(int valor)

{

contador++;//Contador que lleva el registro de ediciones del atributo1

atributo1 = valor;//Establecemos el valor del atributo

}

public int getAtributo1()

{

return atributo1;//Retornamos el valor actual del atributo

}

//Get para el contador

public int getContador()

{

return contador;

}

//Notar que no ponemos un set, pues no nos interesa que el contador pueda ser cambiado.

}

## El modificador por defecto (default)

Java nos da la opción de no usar un modificador de acceso y al no hacerlo, el elemento tendrá un acceso conocido como defaulto acceso por defecto que permite que tanto la propia clase como las clases del mismo paquete accedan a dichos componentes (de aquí la importancia de declararle siempre un paquete a nuestras clases).

package aap.ejemplo2;

public class Ejemplo2

{

private static int atributo1;//Este atributo es privado

static int contador = 0; //Contador con acceso por defecto

public static void setAtributo1(int valor)

{

contador++;//Contador que lleva el registro de ediciones del atributo1

atributo1 = valor;//Establecemos el valor del atributo

}

public static int getAtributo1()

{

return atributo1;//Retornamos el valor actual del atributo

}

}

package aap.ejemplo2;

public class Ejemplo2\_1

{

public static int getContador()

{

return Ejemplo2.contador;//Accedemos directamente al contador desde otra clase

}

}

package aap.ejemplo3;

public class Ejemplo3

{

protected static int atributo1;//Atributo protected

private static int atributo2; //Atributo privado

int atributo3;//Atributo por default

public static int getAtributo2()

{

return atributo2;

}

}

package aap.ejemplo3\_1;

import aap.ejemplo3.Ejemplo3;//Es necesario importar la clase del ejemplo 3

public class Ejemplo3\_1 extends Ejemplo3

{

public static void main(String[] args)

{

//La siguientes dos líneas generan error, pues atributo2 es privado y atributo 3 es default

//System.out.println(atributo2);

//System.out.println(atributo3);

System.out.println(atributo1);//Sí tenemos acceso a atributo1

}

}

## Modificador public

El modificador de acceso public es el más permisivo de todos, básicamente public es lo contrario a private en todos los aspectos (lógicamente), esto quiere decir que si un componente de una clase es public, tendremos acceso a él desde cualquier clase o instancia sin importar el paquete o procedencia de ésta.

A continuación y ya para finalizar, pondré una pequeña tabla que resume el funcionamiento de los modificadores de acceso en Java.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modificador | La misma clase | Mismo paquete | Subclase | Otro paquete |
| private | Sí | No | No | No |
| default | Sí | Sí | No | No |
| protected | Sí | Sí | Sí/No | No |
| public | Sí | Sí | Sí | Sí |

Debo aclarar algo en el caso del modificador de acceso protected y el acceso desde suclases. Es un error común pensar que se puede crear un objeto de la clase madre y luego acceder al atributo con acceso protected sin problemas, sin embargo esto no es cierto, puesto que el modificador protected lo que nos permite es acceder al atributo heredado desde el ámbito de la clase hija y no directamente