**¿Por qué debo aprender Kotlin?**

**Kotlin** es uno de los lenguajes oficiales que **Google** ha adoptado como parte de la familia de lenguajes de programación para desarrollar con **Android**. Kotlin fue desarrollado desde el año 2010 gracias a JetBrains y su primera versión fue liberada en el año 2016.

**Kotlin** es:

* **Conciso**: reduce código a diferencia de Java.
* **Seguro**: Kotlin tiene un método para evitar las excepciones así como para manejar las *NullPointerException*.
* **Interoperable**: **Kotlin** al ser creado por **JetBrains** lo decidieron hacer basado en la máquina virtual de Java, es decir puedes trabajar con **Java** y **Kotlin** en una aplicación ya que comparten la misma máquina virtual.
* **Versátil**: podemos aplicarlo en diferentes tipos de aplicaciones. Del lado del servidor con *Kotlin Server Side - Ktor*, del lado *Mobile Android* y del lado web con *KotlinJS*.

**Java** usa *javac* y **Kotlin** usa *kotlinc* ambos al ser compilado se transforman en código **ByteCode** que es el código que lee la máquina virtual de **Java** (*Java Virtual Machine*)

Kotlin/Native for Native es una tegnologia que complila codigo binario nativo, es decir que podemos hacer apliaciones para todas las plataformas

* iOS (arm32, arm64, simulator x86\_64)
* MacOS (x86\_64)
* Android (arm32, arm64)
* Windows (mingw x86\_64, x86)
* Linux (x86\_64, arm32, MIPS, MIPS little endian, Raspberry Pi)
* WebAssembly (wasm32)

**Mi primer programa en Kotlin**

Para empezar a configurar nuestro entorno de desarrollo necesitaremos varias herramientas:

* El SDK de Java ya que **Kotlin** está basado en **Java**
* IntelliJ IDEA el cual será nuestro editor de código el cual ya tiene un compilador de código.

A continuación te presentamos un *Hola Mundo* en Kotlin:

**fun** **main** (args: **Array<String>**) {

println("Hola Platzi")

}

# Variables vs. Objetos

En **Kotlin** variables vs. objetos significa otra cosa en comparación a otros lenguajes de programación.

Una **variable** es un espacio en memoria que nosotros reservamos para almacenar un solo dato.

Un **objeto** es un espacio en memoria que es más complejo que una variable, se va a componer de otras variables y otros objetos, acciones, métodos y funcionalidades.

En **Kotlin** todo será un objeto, evitaremos el tipo de datos variables, simples o sencillos (tipos primitivos). No debemos declarar valores primitivos (si podríamos pero no debemos), pero son definidos cuando no los usamos como objetos.

Ejemplo de dato **primitivo**:

**var** i = 10

i = i \* 2

println(i)

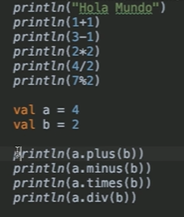
**Kotlin** utiliza wrappers (clases envoltorio) para los números, esto se llama boxing.

En los operadores de **Kotlin** debemos tratar todo como si fuera un objeto:

| **Expresión** | **Función** | **Se traduce a** |
| --- | --- | --- |
| a + b | plus | a.plus(b) |
| a - b | minus | a.minus(b) |
| a \* b | times | a.times(b) |
| a / b | div | a.div(b) |
| a % b | mod | a.mod(b) |
| a += b | a = a + b | a.plusAssign(b) |
| a -= b | a = a - b | a.minusAssign(b) |
| a \*= b | a = a \* b | a.timesAssign(b) |
| a /= b | a = a / b | a.divAssign(b) |
| a %= b | a = a % b | a.modAssign(b) |

| **Operador** | **Significado** | **Expresión** | **Se traduce a** |
| --- | --- | --- | --- |
| + | Suma unaria | +a | a.unaryPlus() |
| - | Resta unaria | -a | a.unaryMinus() |
| ! | Negación | !a | a.not() |
| ++ | Incremento | ++a | a.inc() |
| -- | Decremento | --a | a.dec() |

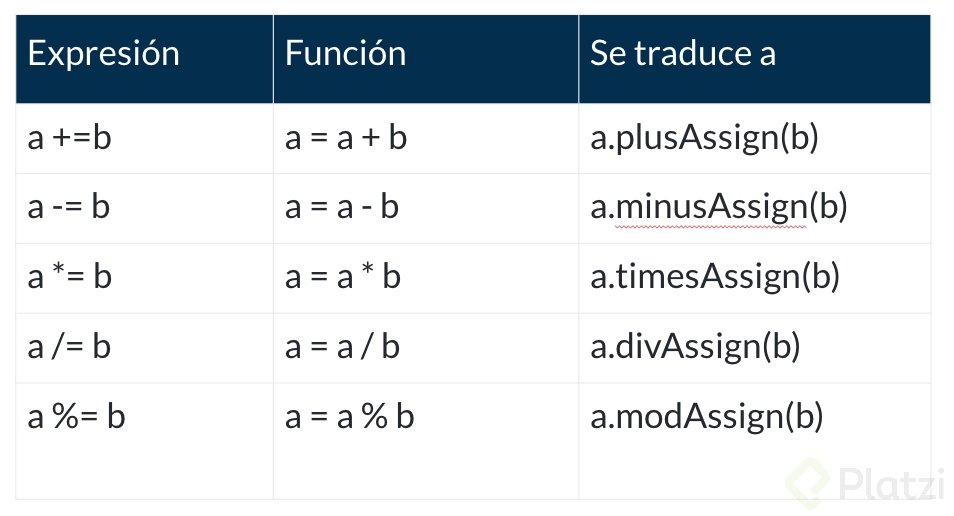
# Operadores en Kotlin



# Otros Operadores en Kotlin

Cuando usamos un operador en Kotlin él compilador lo traduce a una función específica.  
En la tabla que estás mirando te muestro la traducción que usa el compilador para ejecutar la operación.

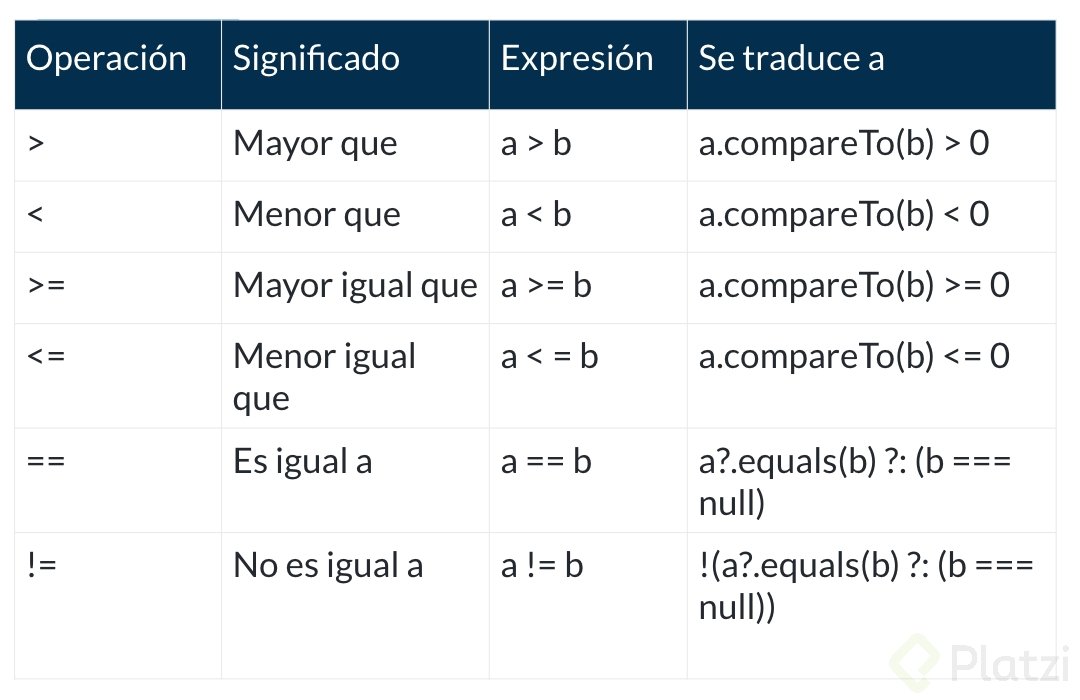
## Asignación



## Unarios, Incremento y Decremento



## Equidad



Hagamos algunos ejemplos para entender cómo funcionan.

Los de asignación son sencillos de entender, basta con la explicación y el desglose que se proporciona en la tabla.

**Los operadores unarios** no harán más que seguir y tratar de aplicar la regla de los signos, en este caso puedes aplicar el operador o utilizar el método al cual Kotlin lo traduce. Algo así:

**var** **a** = -2

**var** **b** = **a**.unaryMinus()

**println**("b: $b")

Esto imprimirá:  
b: 2

Lo que hace es convertir el dato a positivo, pues siguiendo la regla de los signos (-) por (-) es (+).

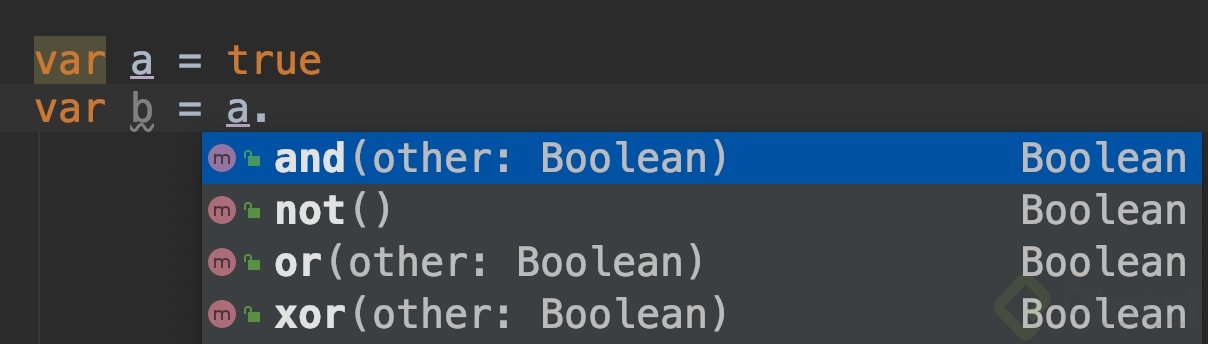
El operador de negación, funcionará con datos lógicos, este simplemente negará el dato establecido.

**var** **a** = true

**var** **b** = **a**.not()

**println**("b: $b")

El resultado será:  
b: false

Al utilizar datos lógicos, la función not() no será la única que podemos usar, tenemos otras opciones como se muestra en la figura:  


**El operador incremento y decremento.** Este se dedicará a incrementar o decrementar el valor de la variable en 1. Aquí ocurrirán dos momentos en que se puede incrementar o decrementar la variable, estos se llamarán: Pre-Incrementar a o Post-Incrementar a y Pre-Decrementar --a y Post-Decrementar a–.

Cuéntame en la sección de discusiones cuál es la diferencia entre usar Pre o Post.

Los operadores de equidad se enfocarán en comparar si un dato es igual, o mayor, o mayor e igual que otro, etc. su resultado será un valor lógico. Estos pueden ser utilizados en los controladores de flujo como if, when, for, while etc.

val **a** = -12

val **b** = 12

**println**("a es mayor que b. ${a>b}")

Reportar un problema

**Tipos de variables: var, val y const**

# Tipos de variables: var, val y const

Hay dos tipos de variables en **Kotlin**: changeables (variables que pueden cambiar) y unchangeables (variables que no pueden cambiar)

Las variables que **pueden cambiar** son declaradas con las palabra reservada var y las variables que **no pueden cambiar** son declaradas con la palabra reservada val.

Las const y las val son prácticamente lo mismo: valores que no se pueden cambiar. La diferencia entre estas dos es: el valor de const se determina en el tiempo de compilación y el valor de val se determina en tiempo de ejecución.

# Programación Funcional: Funciones Puras e Inmutabilidad

[**anncode**](https://platzi.com/@anncode)

7 de Abril de 2019

Comencemos nuestro primer encuentro con la programación funcional. Primero entendamos qué es.

Como primera declaración debemos decir que la **programación funcional es un paradigma de programación**, esto significa, literalmente, un estilo de resolver problemas para plasmar la solución con código. Cuando escuchas la palabra **paradigma** probablemente lo primero que viene a tu mente es el Orientado a Objetos, y sí, precisamente ese es otro estilo de resolución de problemas. Programar de manera funcional significa que lo haremos de forma declarativa, es decir nos preocuparemos más por el **qué** que por el **cómo**.

Este estilo de programación tiene que ver con usar funciones todo el tiempo, podemos usarlas como parámetros de otras funciones e incluso asignarlas a variables. En este paradigma casi todo lo que habíamos aprendido sobre programación dejaremos de usarlo. Nos olvidaremos de usar variables como algo que puede cambiar, en su lugar creeremos en la **inmutabilidad**. Los bucles for, while, do while, etc. dejarán de existir pues ahora tendremos **recursividad**, además que, como ya dijimos, las funciones serán tan especiales que estas deberán ser **funciones puras** por definición.

## Funciones Puras

Una función pura, deberá cumplir con dos cosas específicamente.

**Primero**. Dados los mismos parámetros de entrada la función debe retornar siempre el mismo valor.

**Segundo**. La función no debe tener efectos colaterales, es decir no debe haber nada en el entorno que la altere. Como por ejemplo, variables globales que fue el ejemplo que vimos en la clase anterior.

## Inmutabilidad

Hablando de variables, hace un momento te comenté que estas las dejaremos de tratar como tal y fomentaremos el uso de **variables de tipo constante**, o variables inmutables.

La inmutabilidad es uno de los principios de la programación funcional donde nos promueve la ausencia de estado mutable o también conocido como **Stateless**, para entender esto mejor lo primero que debemos comprender es qué cosa se considera estado.

**Estado**: será cualquier dato que se pueda guardar y modificar posteriormente en memoria

* Una variable
* Un archivo
* Un socket

Para el caso de las variables en Kotlin preferirémos declararlas como ´val´ en vez de ´var´.  
Una buena práctica en nuestros programas es nunca tener variables de tipo ´var´ declaradas de manera **global** pues estas van en contra de la inmutabilidad, aunque sé que a veces es complicado prescindir de ellas en el resto del programa. Su mejor lugar es cuando están declaradas de manera local al método.

A lo largo del curso seguiremos aprendiendo pequeñas píldoras de conocimiento sobre la programación funcional.

Reportar un problema

**Strings**

# Strings

**Raw String**: En Kotlin, además, tenemos cadenas sin procesar de tres comillas que pueden contener caracteres especiales sin la necesidad de escapar de ellas.

**.trimIndent()**: Detecta una sangría mínima común de todas las líneas de entrada, la elimina de cada línea y también elimina la primera y la última si están en blanco.

**.trimMargin()**: Recorta los caracteres de espacio en blanco iniciales seguidos de un marginPrefix de cada línea de una cadena de origen y elimina la primera y la última línea si están en blanco.

Puedes usar los siguientes para escapar caracteres especiales: \t, \b, \n, \r, \', \", \\ y \$

\t : Tabula  
\b : Retrocede el cursor de texto 1 carácter  
\n : Crea un salto de linea  
\r : Mueve el cursor de texto al inicio de la linea en la que está  
\’, ", \ y $ : Te permite usar los caracteres que hay luego del \ dentro de las comillas sin conflictos

Los Strings Template son literales de texto que habilitan el uso de expresiones incrustadas.

**val** nombre = "Kotlin"

**val** apellido: String = "Lang"

print("Tu nombre es: $nombre $apellido")

**val** nacimiento:**Int** = 2011

print("Tu edad es: ${2018-nacimiento}")

**Conversión de tipos de datos**

[**anncode**](https://platzi.com/@anncode)

8 de Abril de 2019

En algunos lenguajes de programación cuando hablamos de tipos de datos y su uso, podría ser correcto asignar un tipo de dato que en memoria tiene menor tamaño a uno que tiene mayor. Por ejemplo un tipo de dato entero a uno de tipo long. En Kotlin esto produciría un error.

Es por eso que en Kotlin tenemos unas funciones dedicadas a la conversión de tipos:

* toByte()
* toShort()
* toInt()
* toLong()
* toFloat()
* toDouble()
* toChar()

Si tuviéramos:

var a: Int = 5

Lo asignaríamos de esta forma:  
Var b: Long = a.toLong()

A continuación te muestro una lista de las posibles conversiones que podemos hacer:

String a Int e Int a String  
Long a Int e Int a Long  
String to Int, and Int to String Conversion  
Double a Int e Int a Double  
Long a Double y Double a Long  
Char a Int e Int a Char  
String a Long y Long a String  
String a Array y Array a String  
String a Boolean y Boolean a String  
String a Byte y Byte a String  
Int a Byte y Byte a Int

# Rangos

**Rango**: nos permite imprimir una secuencia de caracteres.

Ejemplo:

**val** OnToHundred = 1..100

**for** (i **in** OnToHundred){

println(i)

}

Como podemos observar el número se va a imprimir en la consola en un rango de 1 a 100. Podemos hacerlo también con caracteres, como en este ejemplo:

**val** AtoC = 'A'..'C'

**for** (letter **in** AtoC) {

println(letter)

}

También podemos iterar números de forma inversa con la funcion downTo().

**for** (**i** **in** 5 downTo 1){

println(i)

}

**If y when en Kotlin**

Hoy aprenderás sobre sentencias de control, permiten modificar el flujo de ejecución de las instrucciones de un programa.

* **Operador if**: Es uno de nuestros operadores lógicos vistos en clases anteriores, esta sentencia va a funcionar a partir de la comparación de unos datos. El resultado será un valor booleano, true o false.

**val** numero = 2

**if** (numero.equals(2)) { // Utilizando las variables como objetos

println("Si, son iguales")

} **else** {

println("No, no son iguales")

}

* **Operador when**: En **Kotlin** when es parecido al switch de otros lenguajes, usualmente lo veremos acompañado con un rango:

**when**(numero){

**in** 1..5 -> println("Si, está entre 1 y 5")

**in** 1..3 -> println("Si, está entre 1 y 3")

!**in** 5..10 -> println("No, no está entre 5 y 10")

**else** -> println("No está en alguno de los anteriores")

}

# Bucles while y do while

Las diferencias entre usar **while** o **do while** son:

While se utiliza en contadores, es una combinación entre if y for, evalúa una sentencia:

**var** i = 1 // Recuerda que usar var es para variables globales

**while** (i **<= 10){ // Se ejecutará mientras i sea menor o igual a 10, infinitamente**

**println("mensaje $i")**

**i++ // Esto incrementará var hasta que llegue a 10**

**}**

Do while primero va a ejecutar la sentencia y luego la va a evaluar

**var** i = 1

**do** {

println("mensaje $i")

} **while** (i **<= 10)**

# Ciclos for y foreach en Kotlin

[**anncode**](https://platzi.com/@anncode)

8 de Abril de 2019

Nos adelantamos un poco y en clases anteriores aprendimos a usar el ciclo for, como viste Kotlin lo hace muy sencillo de entender pues su sintaxis es muy clara.

Veamos un grado más de complejidad en los ciclos for y además aprendamos sobre foreach.

**Básicamente sabemos que un ciclo for itera o recorre una lista de elementos.**

Imaginemos que tenemos la siguiente lista:

**var** daysOfWeek = listOf("Domingo","Lunes","Martes","Miercoles","Jueves","Viernes","Sabado")

La forma tradicional de iterarlo sería así:

**for**(day **in** daysOfWeek){

println(day)

}

Con esto estaremos imprimiendo día por día. En algún punto del curso platicamos lo útil que a veces suele ser **tener acceso índices**, en Kotlin es posible tener acceso al índice de una lista recorrida en un ciclo for, podemos hacerlo con la siguiente sintaxis:

**for**((index,day) **in** daysOfWeek.withIndex()){

println("$index :$day")

}

De esta forma podemos imprimir el día acompañado de su respectivo índice.

## Foreach

Tenemos una forma de recorrer esto mismo pero con una sintaxis más simplificada, esto es utilizando un foreach, que en esencia tendrá el mismo objetivo que un for natural de Kotlin, recorrer una lista de elementos, solo que la sintaxis aquí es más clara.

Miremos la misma lista de días, recorridas con un foreach:

**var** daysOfWeek = listOf("Domingo","Lunes","Martes","Miercoles","Jueves","Viernes","Sabado")

daysOfWeek.forEach{

println(it)

}

it será el iterador que contiene cada elemento de la lista mientras es recorrida.

Como vemos aparentemente ambas formas de recorrer listas parecen ser igual, sin embargo se ha comprobado que **un ciclo for es 10x más rápido que uno foreach.**

* **Si es para un rango es mejor usar For**
* **Si es para una colección (sequence, list, set) es mejor foreach**
* **Si se va a usar break y continue es mejor For**

# rtículoBreak, Continue y Labels

[**anncode**](https://platzi.com/@anncode)

8 de Abril de 2019

Mientras tomas el control del flujo repitiendo una acción determinadas veces, en algún momento podemos necesitar interrumpirlos cuando una condición se cumpla. Esto se vuelve más útil al tener ciclos anidados, probablemente quisiéramos saltar de un ciclo a otro o simplemente interrumpir las o alguna de las múltiples anidaciones para después continuar con el flujo del programa.

En Kotlin podemos usar **Break y Continue** para lograr esto.

**Break**. Termina el ciclo más cercano

**for** (i **in** 1..3) {

println("\ni: $i ")

**for** (j **in** 1..5) {

**if** (j.equals(3)) break

println("j: $j")

}

}

El resultado es:

i: 1

j: 1

j: 2

i: 2

j: 1

j: 2

i: 3

j: 1

j: 2

**Continue**. Va a la siguiente línea de código del ciclo más cercano.

**for** (i **in** 1..3) {

println("\ni: $i ")

**for** (j **in** 1..5) {

**if** (j.equals(3)) continue

println("j: $j")

}

}

El resultado es:

i: 1

j: 1

j: 2

j: 4

j: 5

i: 2

j: 1

j: 2

j: 4

j: 5

i: 3

j: 1

j: 2

j: 4

j: 5

## Labels

Podemos usar labels para controlar mejor los saltos y **definir en qué ciclo queremos que inicie después de saltar.**  
Un label será un nombre que fungirá como identificador clave para el punto y/o ciclo específico al cual deseamos saltar, lo usaremos con break y continue llamándolo por el mismo nombre definido.

La **sintaxis** de un label es: nombre@  
Un label se **llamará** así: break/continue@nombre

Para entender mejor cómo funciona anidemos un for más a nuestra secuencia:

## Break y Label

**for** (i **in** 1..3) {

println("\ni: $i ")

**for** (j **in** 1..3) {

println("\nj: $j")

**for** (k **in** 1..5) {

**if** (k.equals(3)) break

println("k: $k")

}

}

}

Nos imprime lo siguiente:

i: 1

j: 1

k: 1

k: 2

j: 2

k: 1

k: 2

j: 3

k: 1

k: 2

i: 2

j: 1

k: 1

k: 2

j: 2

k: 1

k: 2

j: 3

k: 1

k: 2

i: 3

j: 1

k: 1

k: 2

j: 2

k: 1

k: 2

j: 3

k: 1

k: 2

Así se verían nuestros for’s usando un label

terminarTodoCiclo@ **for** (i **in** 1..3) {

**println**("\ni: $i ")

**for** (j **in** 1..3) {

**println**("\nj: $j")

**for** (k **in** 1..5) {

**if** (k.equals(3)) **break**@terminarTodoCiclo

**println**("k: $k")

}

}

}

Nos imprime lo siguiente:

i: 1

j: 1

k: 1

k: 2

Cómo ves en este caso para **Break y Label** terminarán el ciclo donde está definido el label.

Ahora veamos cómo se comporta con Continue:

**for** (i **in** 1..3) {

println("\ni: $i ")

**for** (j **in** 1..3) {

println("\nj: $j")

**for** (k **in** 1..5) {

**if** (k.equals(3)) continue

println("k: $k")

}

}

}

Esto es lo que imprime:

i: 1

j: 1

k: 1

k: 2

k: 4

k: 5

j: 2

k: 1

k: 2

k: 4

k: 5

j: 3

k: 1

k: 2

k: 4

k: 5

i: 2

j: 1

k: 1

k: 2

k: 4

k: 5

j: 2

k: 1

k: 2

k: 4

k: 5

j: 3

k: 1

k: 2

k: 4

k: 5

i: 3

j: 1

k: 1

k: 2

k: 4

k: 5

j: 2

k: 1

k: 2

k: 4

k: 5

j: 3

k: 1

k: 2

k: 4

k: 5

Solamente salta y no imprime cuando k=3

## Continue+Label

escaparJ@ **for** (i **in** 1..3) {

**println**("\ni: $i ")

**for** (j **in** 1..3) {

**println**("\nj: $j")

**for** (k **in** 1..5) {

**if** (k.equals(3)) **continue**@escaparJ

**println**("k: $k")

}

}

}

Nos imprime lo siguiente:

i: 1

j: 1

k: 1

k: 2

i: 2

j: 1

k: 1

k: 2

i: 3

j: 1

k: 1

k: 2

Lo que observamos es que **Continue con Label** hace saltar o escapar una iteración entera que en este caso es J.

Usar labels nos ayuda a controlar nuestros ciclos de una forma más eficiente, sin necesidad de utilizar nuestros propios “hacks” de código y obtener los resultados que queremos.  
Las situaciones que pueden presentarse son infinitas y tener este control nos ayuda codificar mucho más rápido.

# Valores Nulos y Double Bang

* Una buena práctica de programación es comenzar con variables **no nulas**.
* Kotlin es **Null Safety**.
* **Kotlin** evita que una excepción sea lanzada porque provoca vulnerabilidades.
* En Kotlin **por defecto ningún valor** puede ser **nulo**. Nos marcará un error.

Podemos declarar valores nulos de la siguiente manera:

**var** variable: **Int**?

variable?.metodo()

// O de esta manera

**var** filePath = arguments?.getString(ARGUMENTS\_PATH)

// Podemos usar Double Bang !!

**var** msg: String?

msg = **null**

println(msg!!.length)

El operador **Double Bang** se utiliza pocas veces en Kotlin y es preferible **no usarlo**

# Operador Elvis

El **operador Elvis** ?: es definido de esta manera debido a que el signo de interrogación asemeja el cabello de Elvis Presley.

Para comparar si un valor es nulo podemos hacer lo siguiente:

**if** (list != **null**) **return** list.size

**else** **return** 0

Con Elvis podemos hacer lo siguiente:

it.listFiles()?.size ?: 0

Los dos dan el mismo resultado, pero con Elvis podemos simplificarlo mucho más.

**Ejercicios con Null Safety y operador Elvis (Clase 12; Curso Basico de Kotlin)**

Si vas a ocupar una función \*\* double Bang;\*\* siempre tiene que ir con su **catch** y así no será vulnerable a los usuarios maliciosos,

**Llamada segura;** esta es ocupada con su icónico signo ? y sirve para declarar a una variable nula.

**Operador Elvis;** el operador Elvis cuenta con muchos códigos como él;filterNotNull; que sirve para excluir a los valores nulos de los otros valores, este también puede acortar el código de una asombrosa manera.

Les comparto unos [Ejerccios](https://github.com/JandresR86/ejercicios" \t "_blank)  
para practicar, recuerden que la practica hace al maestro

# ¿Qué es un Array en Kotlin?

Los **Array** son colecciones de datos que nos permiten almacenar grandes o múltiples cantidades de información en una sola variable.

-ArrayOf(): abrimos la posibilidad para que no sea un tipo de dato específico, para cualquier tipo de datos.

Podemos especificar un tipo de datos, por ejemplo de tipo entero:

**val** numbers = intArrayOf(1, 2, 3, 4)

Para **iterar** un arreglo e imprimirlo en la pantalla podemos hacerlo de la siguiente manera:

**for** (num **in** numbers){

println("Numeros: $num")

}

**fun** arreglos(){  
 *//Array de multivalor* **val** countries = *arrayOf*(**"Indía"**, **"México"**, **"Chile"**, 4, 10 ,2.5)  
 *//Array con restición* **val** days: Array<String> = *arrayOf*<String>(**"Lunes"**, **"Martes"**, **"Miercoles"**)  
 *//Arreglo especifico* **val** number = *doubleArrayOf*(2.5, 20.34, 18.19 )  
  
 *//Formas de recorrer array  
 //For* **for** (countrie **in** countries){  
 *println*(countrie)  
 }  
 *//ForEach* days.*forEach* **{** *println*(**it**)  
 **}** *//Promedio de números* **var** suma:Double = 0.0  
 **for**(num **in** number){  
 suma +=num;  
 }  
 **val** avergege = suma / number.**size**;  
 *println*(**"Promedio arreglo double es de: $**avergege**"**)  
}

Este código

**val** numbers:IntArray = intArrayOf(1,3,5,7,9)

**var** suma = 0

**for** (number **in** numbers) suma = suma.plus(number)

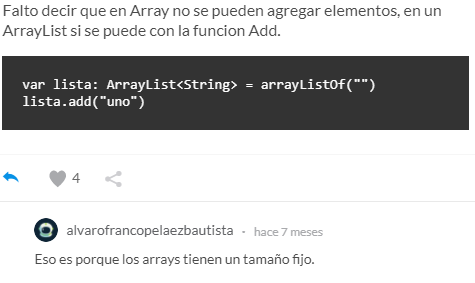
**val** average = suma.div(numbers.size)

println("Promedio de números ${average}")

Es igual a

val numbers:IntArray = intArrayOf(1,3,5,7,9)

**println**("Promedio de números ${numbers.average()}")



# Métodos útiles en arreglos

Cuando escribimos arrayOf estamos preparando el arreglo para manejar objetos; pero cuando escribimos intArrayOf es para manejar tipos de datos primitivos.

Cuando queremos pasar tipos de datos de objeto a datos primitivos podemos hacerlo de la siguiente manera:

**var** arrayObject = arrayOf(1, 2, 3, 4)

**var** intPrimitive : IntArray = arrayObject.toIntArray()

// Podemos hacer una suma sin necesidad de un bucle de esta manera

**val** suma = arrayObject.sum()

println("La suma del array es: $suma")

# Métodos útiles en arreglos

Cuando escribimos arrayOf estamos preparando el arreglo para manejar objetos; pero cuando escribimos intArrayOf es para manejar tipos de datos primitivos.

Cuando queremos pasar tipos de datos de objeto a datos primitivos podemos hacerlo de la siguiente manera:

**var** arrayObject = arrayOf(1, 2, 3, 4)

**var** intPrimitive : IntArray = arrayObject.toIntArray()

// Podemos hacer una suma sin necesidad de un bucle de esta manera

**val** suma = arrayObject.sum()

println("La suma del array es: $suma")

**fun** arreglosUtil(){  
  
 **var** arrayObject = *arrayOf*(5,3,7)  
 *//Transfroma el arreglo a primitive* **var** intPrimitive : IntArray = arrayObject.*toIntArray*()  
 *//Metodos en arreglos* **val** suma = arrayObject.*sum*();  
 *println*(**"El valor de la suma del arreglo tipo obj $**suma**"**)  
 *//Agregar datos a un arreglo* arrayObject = arrayObject.*plus*(12)  
 arrayObject.*forEach* **{** *print*(**"$it,"**)  
 **}** *println*(**""**)  
 *println*(**"reversedArray"**)  
 *//Invertir datos de un arreglo* arrayObject = arrayObject.*reversedArray*()  
  
 arrayObject.*forEach* **{** *print*(**"$it,"**)  
 **}  
  
 val** suma2 = intPrimitive.*sum*()  
  
  
  
}

# Expresiones vs. Valores

En **Kotlin** todo es una expresión. Siempre se debe devolver un valor aún si ese valor es nada.

Las **expresiones** se definen como el conjunto de variable y operadores que devuelven un valor.

${} -> colocar expresiones

$ -> colocar valores

Algunas excepciones que no pueden ser una expresión son: **Bucles for y while**

**fun** expresiones(){  
 **var** x = 5  
 *//Expreción  
 println*(**"X es igual a 5? ${**x==5**}"**) *// true* **var** mensaje = **"El valor de x es $**x**"** x++  
 *println*(**"${**mensaje.*replace*(**"es"**, **"fue"**)**}, x es igual a: $**x**"**)  
}

# Funciones en Kotlin

En **Kotlin** tenemos dos tipos de funciones:

* Funciones provistas por Kotlin
* Funciones declaradas por ti.

**En Kotlin siempre se devuelve un valor**

Todas las funciones en Kotlin comenzarán con la palabra reservada **fun**

Una función se llamará de la siguiente forma: platzi("Hola mundo"). El nombre de la función seguido de los paréntesis y si se necesita colocaremos los parámetros entre ellos.

**Unit** es la forma de decir que una función no devuelve nada-

**fun** funcion(){  
 *println*(**"Raiz cuadrade de ${**Math.sqrt(4.0)**}"**)  
  
 **fun** averageNumbers(numbers: IntArray): Int {  
 **val** sumas = numbers.*sum*()  
 **return** sumas.div(numbers.**size**)  
 }  
 **val** numeros = *intArrayOf*(8, 7, 20)  
  
 *println*(**"El promedio de los numeros 8, 7 y 20 es ${**averageNumbers(numeros)**}"**)  
}

# Lambdas en Kotlin

**Lambdas**: soluciona una nueva forma de escribir funciones. En otros lenguajes se conocen como funciones anónimas, functions literals o funciones literales. Decimos que son funciones anónimas porque no tienen nombre.

Las **lambdas** son una expresión que hace una función.

{println("Hola Platzi")}()

**var** hola = {println("Hola Platzi")}()

hola()

**fun** lambdas(){  
 **val** lambdaf = **{**a:Int, c:Int **->** a+c **}** *println*(lambdaf(24,6))  
  
 *//Iniciaizada* **val** lambdaIn = (**{**a:Int, c:Int **->** a+c **}**(1,1))  
  
 *println*(lambdaIn)  
}

La primera ejecuta la función y almacena el valor retornado y la segunda almacena la función en sí sin ejecutar. Lo pueden corroborar de la siguiente manera.

**fun** **random**(): **Int** {

**return** (Math.random() \* 100).toInt()

}

**val** random1 = random()

**val** random2 = {random()}

println(random1)

println(random2)

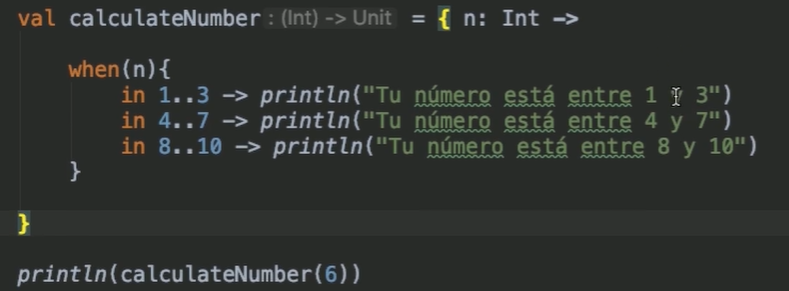
El resultado en la consola será el siguiente

98

() -> kotlin.Int

El primer resultado en el número aleatorio el segundo es una función que no está ejecutada aún. Para ejecutarla habría que hacer:

println(random2())



# Clases

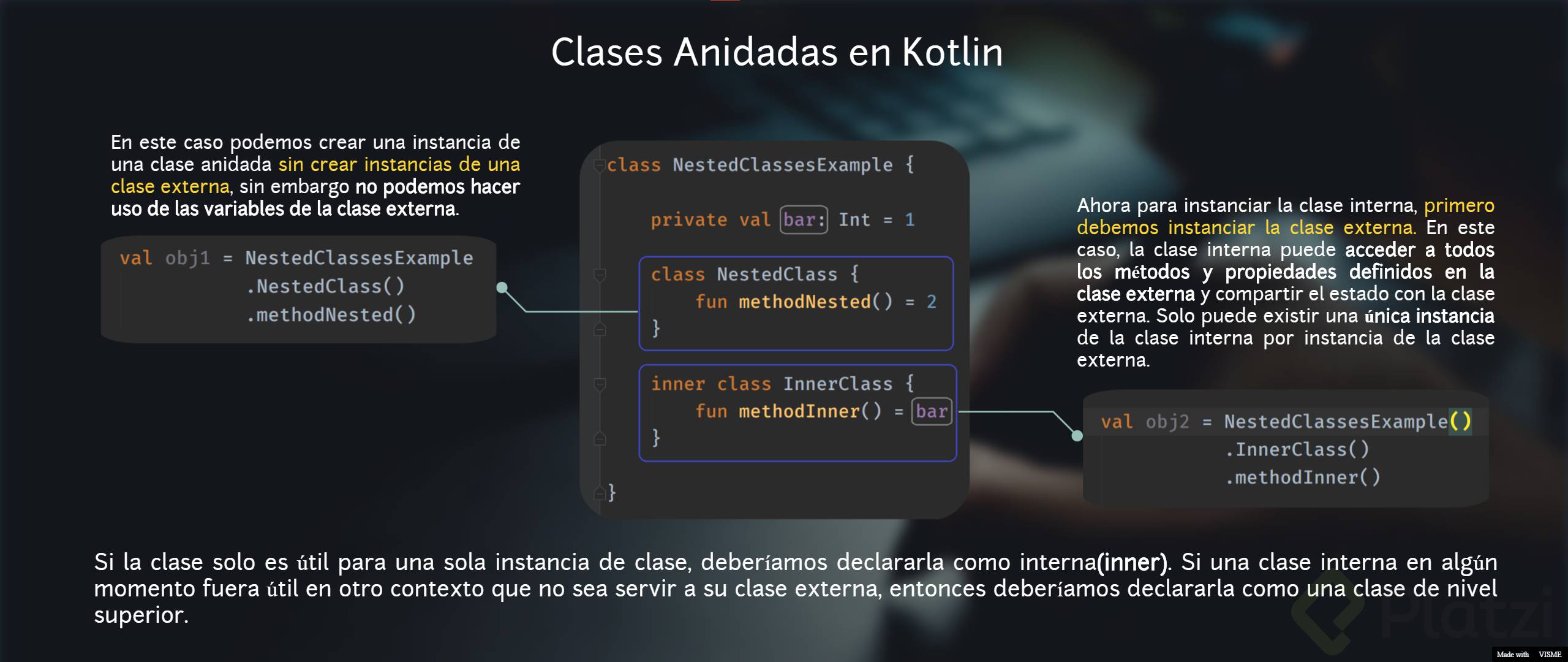
Las **clases** son moldes o templates que permiten crear objetos con multitud de características y funcionalidades.

Normalmente en la programación orientada a objetos entendemos que los objetos son una entidad, por ejemplo una laptop, un móvil, una pantalla, entre otros.

Para crear nuestros objetos es importante partir de **clases**, los objetos tendrán multitud de características, por ejemplo tu teclado no será el mismo de otra persona ya que hay variedad de teclados en el mundo.

Las **clases** nos permiten general un molde genérico para tener multitud de entidades. Para declarar una clase utilizamos la palabra reservada class, por ejemplo para colocar el nombre de una clase lo haríamos de esta manera: class HolaPlatzi.

Hubieron algunos temas que no se tocaron como clases internas(inner class), clases selladas(sealed class), entre otros.  
Les comparto una infografía sobre clases internas que hice para reforzar un poco el aprendizaje (:  
Espero les sea de ayuda, saludos!



# Ejercicio de clases

En resumen…  
Las **Clases** se conocen y se diferencian por:  
• Los nombres de las clases; deben de ser el mismo que el del archivo.  
• Empiezan con mayúscula; los nombres de las clases siempre empiezan con la letra en mayúscula.  
• Los métodos; toda clase que lleve buenas practicas también lleva su métodos.  
• Las expresiones; toda clase que desee se imprimida debe de darse a declarar como expresión.  
Definiciones dentro de las **Clases:**  
• **POO :** Los archivos deben estar separados lo más posible; la Programación Orientada a Objetos (POO) nos dice que los problemas deben estar separados en diferentes módulos, diferentes archivos.

Clase de cámara

**package** model  
  
**class** Camera{  
 **var isOn**: Boolean = **false  
  
 fun** turnOn(){  
 **isOn** = **true** }  
  
 **fun** turnOff(){  
 **isOn** = **false** }  
}

Uso de la clase e instancia

**fun** clases(){  
 **val** camera = Camera()  
 camera.turnOn()  
 *println*(**"La capara eatá: ${**camera.**isOn}"**)  
}

**Modificadores de acceso**

En **Kotlin** todos los valores son públicos por defecto.

**Modificadores de acceso**:

* **public**: todo acceso. Cualquier persona desde cualquier clase, método o lugar puede modificar el dato de esta variable.
* **private**: acceso solo dentro de la clase. Es uno de los modificadores más duros y con mayor restricciones.
* **protected**: acceso solo dentro de la clase y las clases que hereden.
* **internal**: acceso entre módulos.

// Private

**private** **var** size // Restringiendo el acceso o visibilidad; encapsulamiento

**Getters y Setters**

Los **Getters** y **Setters** son métodos de acceso, lo que indica que son siempre declarados públicos, y nos sirven para dos cosas:

* **Getters**: significa obtener, pues nos sirve para obtener (recuperar o acceder) el valor ya asignado a un atributo y utilizarlo para cierto método.
* **Setters**: significa establecer, pues nos sirve para asignar un valor inicial a un atributo, pero de forma explícita. Además el Setter nunca retorna nada (Siempre es *void*) y solo nos permite dar acceso público a ciertos atributos que deseemos el usuario pueda modificar.

**private var resolution**: Int = 640  
  
*//Get y set para variables privadas***fun** setResolution(resolution: Int){  
 **this**.**resolution** = resolution  
}  
  
**fun** getResolution(): Int{  
 **return this**.**resolution**}

Uso

camera.setResolution(1080)  
*println*(**"Resolution: ${**camera.getResolution()**}"**)

# El Get() y Set() propio de Kotlin

Mi resumen sobre Fiel 😄  
• **Field**; es un tipo de variable que toma temporalmente la propiedad size para que pueda ser alterado.

De esta forma podemos realizar validaciones dentro de nuestras clases

Clase

**class** Shoe {   
 **var size**: Int = 34 *// Min 34* **set**(value) {  
 **if** (value >= 34)  
 **field** = value  
 **else field** = 34  
 }  
 **get**() = **field  
 var color**: String = **"White"** *//Cool* **var model**: String = **"Boots"** *// Not Tenis* **set**(value) {  
 **if** (value == **"Tenis"**){  
 **field** = **"Boots"** }  
 **else field** = value  
 }  
  
}

Uso de la clase

**fun** classRestricciones(){  
 **var** shoe = Shoe()  
 shoe.**size** = 39  
 *println*(shoe.**size**)  
 shoe.**model** = **"Zapato"** *println*(shoe.**model**)  
}

**Data class en Kotlin**

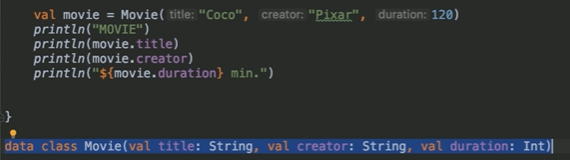
Con frecuencia creamos clases cuyo propósito principal es mantener los datos. En una clase de este tipo, algunas funciones de funcionalidad y utilidad estándar a menudo se derivan mecánicamente de los datos. En Kotlin, esto se llama una data class y se marca como datos:

data **class** **User**(**val** name: String, **val** age: **Int**)

El compilador deriva automáticamente los siguientes miembros de todas las propiedades declaradas en el constructor primario:

* equals()/hashCode()
* toString() de la forma "User(name=John, age=42)"
* componentN() functions correspondiente a las propiedades en su orden de declaración.
* copy()

También es posible crear una data class en uso por ejemplo dentro del Main



**Método Constructor**

Existen dos tipos de constructores en Kotlin:

* **Constructor Primario**: será la forma clásica de inicializar una clase, ya usamos esta forma de constructor en la clase anterior con el data class

**class** Shoe(**var sku**: Int, **var mark**:String)

* **Constructor Secundario**: cuando colocamos más código para inicializar una clase.

Para hacer uso del constructor secundario se hace uso de la siguiente palabra reservada, esté método se ejecuta al momento de crear la clase

**init** {  
 *println*(**"SKU ID $sku"**)  
 *println*(**"Marca $mark"**)  
}