

.dradle, .idea (el . piunto) guarda toda la información de nuestro proyecto

Build: encuentra todo el código compilado previamente por gradle

Gradle: programa que nos permite gestionar las dependencias de nuestro proyecto y compilarlo

Src: dodne se encuentra nuestro código, nuestros archivos .kt

Main: punto de entrada para el desarrollo de nuestra aplicación.

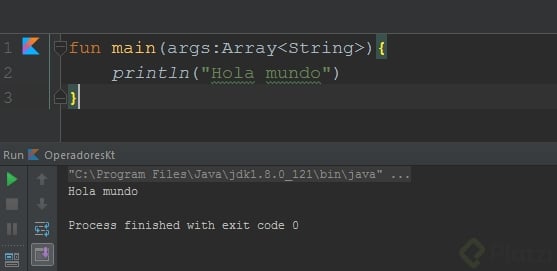
Test: todos los test de nuestra aplicación .

Build.gradle.kts: configuración de cómo funciona nuestro proyecto, dependencias, de obtención de información, versiones de kotlin.

Gradle.propeties: declarar proppiedades para nuestro proyecto, como estilos.

Gradles y gradlew.bat: Plataformas windows y Unix

Settings.gradle.kts: especifica las propiedades de proyecto como en nombre del proyecto.



Los programas en kotlin empiezan con una funcion principal, esta función es el punto de entrada de nuestras aplicaciones, siempre que necesitemos que un programa tenga vida necesitamos crear o tener dicha funcion (fun main).

Está comprendida por:

fun main(args: Array<String>) {

println("Hola Mundo")

}

fun main ----> la funcion principal

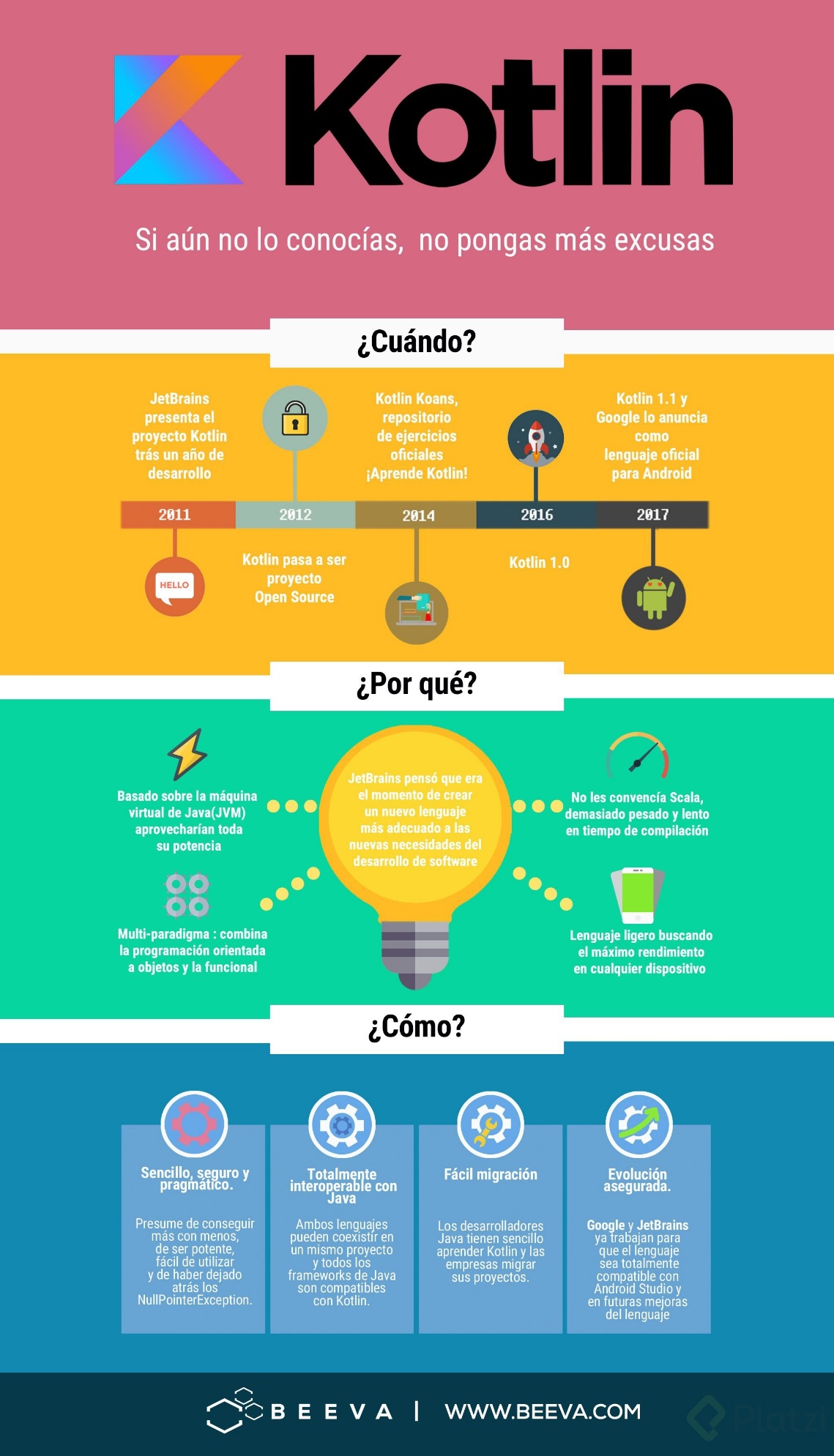
(args: Array<String>) ---> parametros de la función

{} ----> dentro de las llaves se encuentra en codigo que se ejecutará en nuestra aplicación.

print("Hola Mundo") ----> con esta funcion podremos codigo por pantalla.

println("Hola Mundo") ----> es lo mismo que print pero con esta podremos hacer salto de lineas.

Otra cosa que debemos tener en cuenta es que Kotlin es CASE SENSITIVE, esto que quiere decir, que las funciones, palabras reservadas, variable, declaraciones, etc, se deben escribir tal como nos indica el lenguaje y distingue entre minúsculas y mayúsculas.



**Variables en Kotlin**

VARIABLES

Sirve para almacenar datos temporales y utilizarlos a lo largo de nuestro código.. En kotlin las variables solo pueden almacenar un solo dato.

TIPOS DE VARIABLES

VAR : Son variables de lectura y escritura, estas variables el valor puede cambiarse.

VAL : Son variables de solo lectura, dichas variables una vez asignado el valor no puede ser cambiado posteriormente.

CONST : es una variante de las variables de solo lectura, estas se definen fuera de la función y se escriben con la palabra reservada const seguida de la palabra reservada val, este tipo de variables son usada para valores que nunca cambian.

En Kotlin no se puede cambiar el tipo de dato con que se a definido una variable, si la variable se definio con el tipo de dato String solo podremos actualizar dicho valor por otro String, por ejemplo no podemos pasar de un string a un numero.

//const es una variante de las variables de solo lectura, estas se definen fuera de la función y se escriben con la palabra  
reservada const seguida de la palabra reservada val, este tipo de variables son usada para valores que nunca cambian.  
const val *PI* = **3.1416**fun main(args: Array<String>) {  
 *println*("Hello World!")  
 // var: Son variables de lectura y escritura, estas variables el

valor puede cambiarse  
 //variabele dinero que tiene unalor asignado de 10  
 //var dinero : Int = 10  
 var dinero = **10** *println*(dinero)  
 dinero = **5** *println*(dinero)  
 // val: Son variables de solo lectura, dichas variables una vez

asignado el valor no puede ser cambiado posteriormente.  
 val nombre = "Maria"  
 *println*(nombre)  
 *println*(*PI*)  
}

Hello World!

10

5

Maria

3.1416

# Kotlin y sus tipos de variables

🟠Siempre que creamos una variable o una constante esta tiene que estar asociada a un tipo de dato, un tipo de dato es un atributo y dicho atributo lo que va a definir en nuestra variable es tipo de dato, es decir, el tipo de valor que puede tener la variable o contante en nuestro programa.

🟠En Kotlin existe una característica llamada inferencia de tipo, con dicha característica kotlin puede saber a que tipo de dato nos estamos refiriendo con el simple hecho de como colocamos los datos.

Ejemplo:  
val nombre: String = “Ivan” —> Especificando el tipo de dato.

val nombre = “Ivan” ----> Inferencia de tipo (Kotlin sabe que es un string).

**TIPOS DE DATOS**

🔷 STRING: son cadenas de textos.  
Podemos concatenar string utilizando template string, se utilizan comillas dobles.  
“hola me llamo $nombre $apellido”

🔷 ENTEROS: estos se dividen en cuatro (Byte, Short, Int, Long), la única diferencia que existe entre ellos es la longitud del numero que se escriba.

🔸 BYTE: solo podremos representar números entre -127 a 128 (Para enteros de hasta 8bits).

🔸SHORT: solo se pueden representar números enteros de hasta 16bits.

🔸LONG: tiene soporte para una cantidad muy grande de números.

🔸 INT: tipo que representa un número entero.

🔷DECIMALES: tenemos dos tipos de datos decimales (Double, Float), al igual en el caso de los enteros, la diferencia entre estos dos datos es la capacidad de poder trabajar con diferentes longitudes de números.

🔸FLOAT: representa un numero decimal de hasta 32bits.

🔸DOUBLE: representa un numero decimal de 64bits.

🔷BOLEAN (BOOL): es un tipo de dato lógico, con este tipo de dato solo podremos escribir dos condiciones (True, False), las operaciones que podremos realizar con este dato tienen que estar relacionados con lógica.

fun main(args: Array<String>) {  
  
 //val boolean : Boolean = true  
 val boolean = true  
 // long  
 val numeroLargo = **3L** // flotantes  
 val double = **2.7182** // float  
 val float = **1.1f** val primerValor = **30** // el punto. le da superpoderes que peudes ver y hacer lo que necesite  
 //primerValor.and()  
 val segundoValor = **10** val tercerValor = primerValor.minus(segundoValor)  
 *println*(tercerValor)  
 val tercerValor2 = primerValor - segundoValor  
 *println*(tercerValor2)  
  
 val apellido = "ricaurte"  
 val sebas = "sebas"  
 //val nombreCompreto = giuseppe + " " + apellido  
 val nombreCompreto = "$sebas $apellido"  
 *println*(nombreCompreto)  
 val nombreCompleto2 = "Mi nombre es $sebas $apellido "  
 *println*(nombreCompleto2)  
  
}

20

20

sebas ricaurte

Mi nombre es sebas Ricaurte

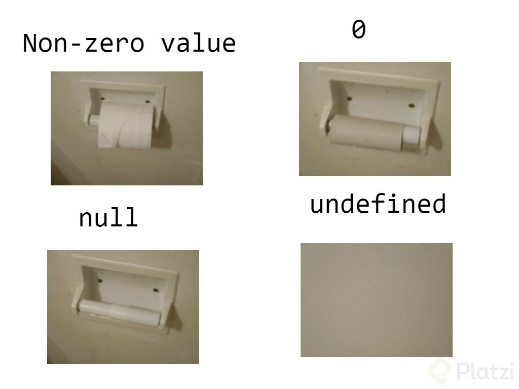
# Modificadores y tipos de datos en Kotlin

**¿Qué es un dato primitivo**  
Tipos de datos originales de un lenguaje de programación. En Kotlin lo son los enteros, booleanos y cadenas de texto

**¿Qué es un objeto**  
Es una combinación de variables, funciones y otros objetos.  
En Kotlin todo es un objeto, se convierten los datos primitivos a un objeto para obtener algunas ventajas como:

* Crear funciones especificas para el objeto que ayuden a no reescribir el código
* Sobrescribir operadores como la suma o multiplicación

Cuando queremos entender los tipos de datos “Nulos” este meme me ayudo a comprenderlo un poco mejor, igual cuando hablamos de un tipo “undefined”.



# Operaciones con los tipos de datos en Kotlin

En **Kotlin** las operaciones son traducidas a funciones interiormente por el compilador. La operación val tercerValor = primerValor + segundoValor es lo mismo que decir tercerValor = primerValor.plus(segundoValor).

En la siguiente tabla te voy a dejar las operaciones que vas a poder realizar con los distintos tipos de datos y si te encuentras con alguno que no permita realizar esa operación puedes crearla por tu cuenta. Recuerda que **Kotlin** te permite extender el lenguaje para aprovechar estas funcionalidades.

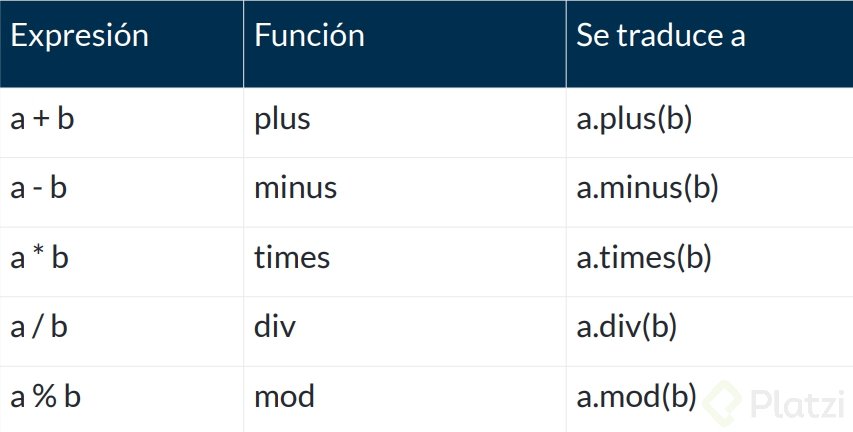
## Operaciones más utilizadas

| **Expresión** | **Función** | **Operator Fun** |
| --- | --- | --- |
| a + b | c = a + b | public operator fun plus(other: Int): Int |
| a - b | c = a - b | public operator fun minus(other: Int): Int |
| a \* b | c = a \* b | public operator fun times(other: Int): Int |
| a / b | a = a / b | public operator fun div(other: Int): Int |
| a % b | c = a % b | public operator fun rem(other: Int): Int |
| a++ | c = a++ | public operator fun inc(): Int |
| a– | c = a– | public operator fun dec(): Int |
| a > b | c = a > b | public override operator fun compareTo(other: Int): Int |
| a < b | c = a < b | public override operator fun compareTo(other: Int): Int |
| a >= b | c = a >= b | public override operator fun compareTo(other: Int): Int |
| a <= b | c = a <= b | public override operator fun compareTo(other: Int): Int |
| a != b | c = a != b | public open operator fun equals(other: Any?): Boolean |

.  
Dependiendo del tipo de dato que tengas podrás utilizar todos o solamente algunas de estas operaciones, por ejemplo si tienes una variable del tipo de dato **String** no vas a poder dividirla, a menos que tú crees esa función. Sin embargo, sí vas a poder sumar dos variables del tipo de dato **String** para obtener el valor de dicha suma.

Con esto espero que hayas obtenido una idea sobre cómo funcionan las operaciones, queda de parte de ti si prefieres utilizar la versión larga del **operator fun** o el operador directamente.

Ten en cuenta que si las **operator fun** se inventaron para que puedas reducir tu código a operaciones con símbolos ¿por algo será, no?









# Estructuras de control: if

fun main(args: Array<String>) {  
 val nombre = "li"  
 val nombre2 = ""  
  
 //codicion, isNotEmpty esta funcion n osayuda indetemina si esta vacio o no, un boleano  
 if (nombre.*isNotEmpty*()){  
 *println*("1.El Largo de nuestro variable nombre es ${nombre.length}") //length propiedad indica el latgo de nombre  
 }else {  
 *println*("Error, la variable esta vacia")  
 }  
 //tambien puede reducir a una sola linea  
 if (nombre2.*isNotEmpty*()) *println*("El Largo de nuestro variable nombre es ${nombre2.length}") else *println*("2.Error, la variable esta vacia")  
  
 val mensaje : String = if (nombre.length > **4**){  
 "tu nombre es largo"  
 } else if(nombre.*isEmpty*()){ // la cadena es igual a cero  
 "Nombre esta vacio"  
 } else {  
 "tienes un nombre corto"  
 }  
 *println*(mensaje)

1.El Largo de nuestro variable nombre es 2

2.Error, la variable esta vacia

tienes un nombre corto

# Estructuras de Control: when

fun main(args: Array<String>) {  
 val nombreColor = "CArmesi"  
  
 when (nombreColor) {  
 "Amarillo" -> *println*("Amarillo color de la alegria")  
 "Rojo"**,**"CArmesi" -> *println*("El color simoliza el calor")  
 else -> *println*("Error. No tengo informacion del color")  
 }  
  
 val code = **501** when (code){  
 in **200**..**299** -> *println*("todo ha ido bien")// in =se encuentra en  
 in **400**..**500** -> *println*("algo ha fallado")  
 else -> *println*("codigo desconocido, algo ha fallado")  
 }  
  
 val tallaDeZapatos = **41** val mensaje = when(tallaDeZapatos){  
 **41,43** -> "tenemos disponibles"  
 **42,44** -> "casi no nos queda"  
 **45** -> "lo siento, no tenemos disponiblidad"  
 else -> "estos zapatos solo viene solo tallas de 41 a 45"  
 }  
 *println*(mensaje)

El color simoliza el calor

codigo desconocido, algo ha fallado

tenemos disponibles

# Bucles: While y Do While

**While :** Vas de Viaje y el Chofer del autobus al Entrar te verefiva el ticket si Cumple las condiciones pasas y así va recorriendo los Viajeros hasta que encuentres un Viajero que no tenga boleto. Se para el Bus y termina

**DoWhile**: Vas de Viaje y el Chofer del autobus al Entrar te saluda y te deja pasar, Luego de eso se levanta y comienza a revisar los ticket si Cumple las condiciones pasas si no Encuentra el Viajero que no cumple y Se termina. Es decir Primero te deja Pasar Y Lugo te Evalua

fun main(args: Array<String>) {  
 var contador = **10** while (contador > **0**){  
 *println*("el valor de ocntador es $contador ")  
 contador--  
 }  
  
 do {  
 *println*()  
 *println*("generando numero aleatorio..")  
 val numeroAleatorio = (**0**..**100**).*random*()  
 *println*("el numero generado es $numeroAleatorio")  
 } while (numeroAleatorio > **50**)  
  
}

el valor de ocntador es 10

el valor de ocntador es 9

el valor de ocntador es 8

el valor de ocntador es 7

el valor de ocntador es 6

el valor de ocntador es 5

el valor de ocntador es 4

el valor de ocntador es 3

el valor de ocntador es 2

el valor de ocntador es 1

generando numero aleatorio..

el numero generado es 64

generando numero aleatorio..

el numero generado es 69

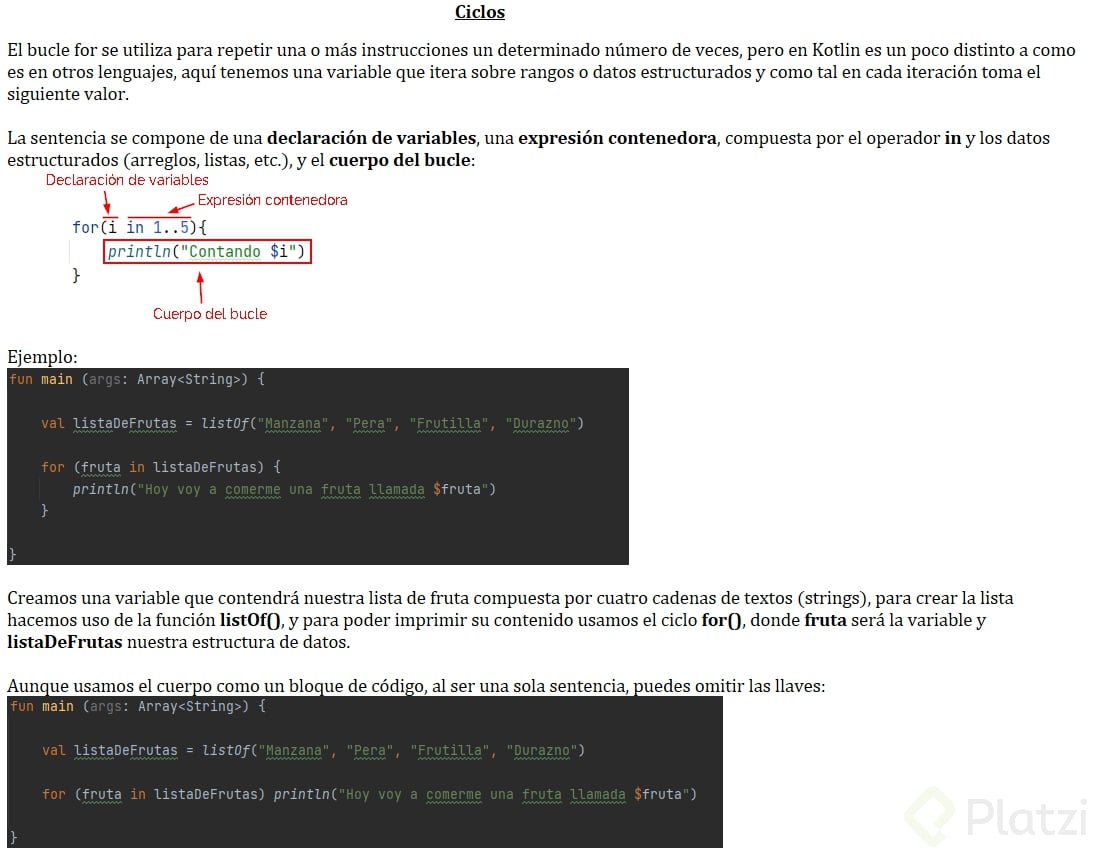
generando numero aleatorio..

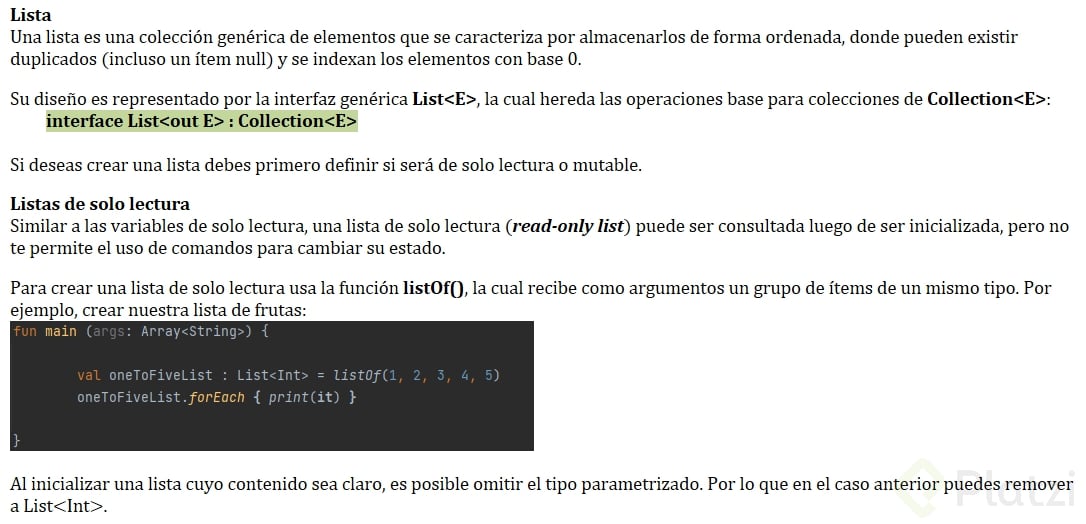
el numero generado es 78

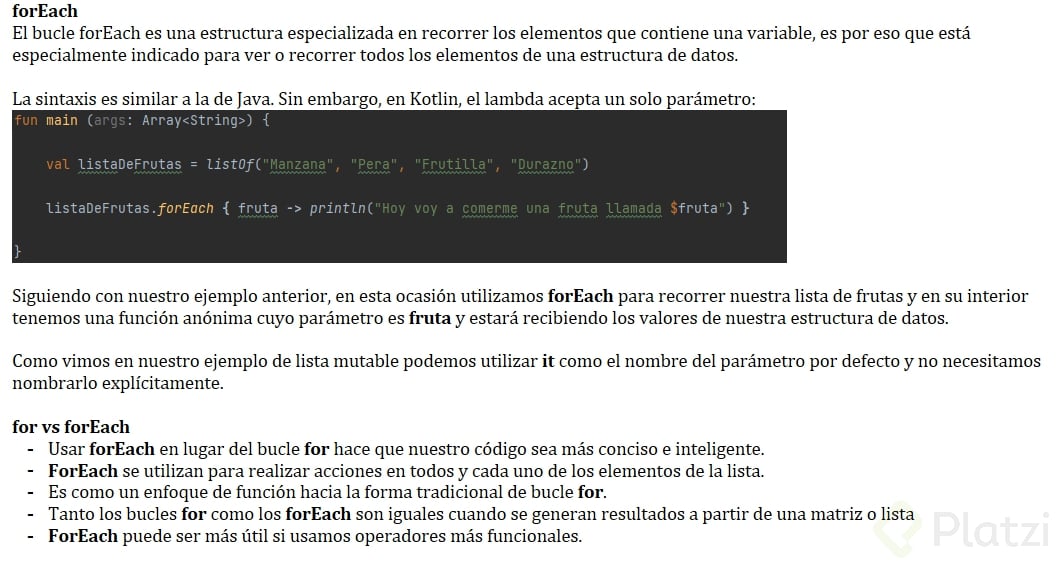
generando numero aleatorio..

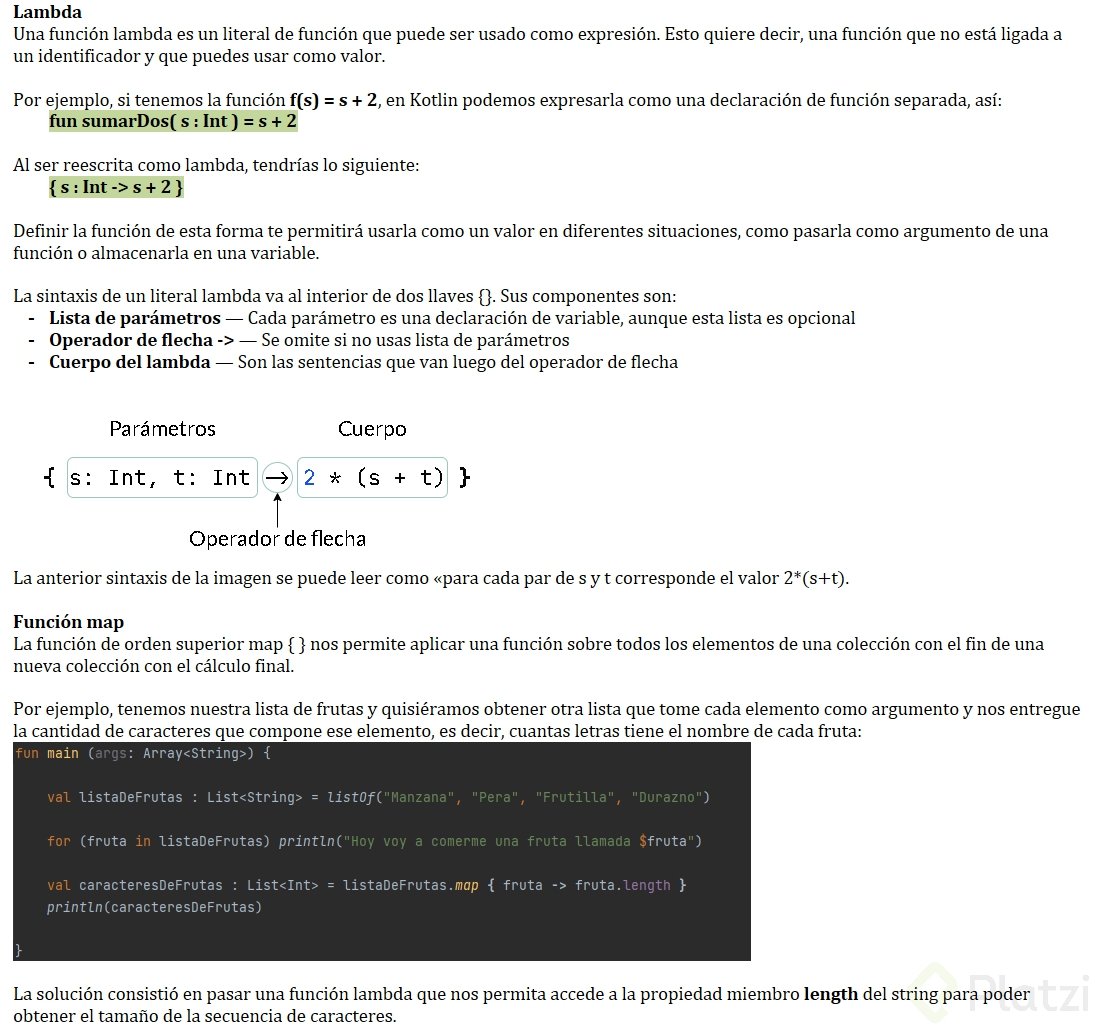
el numero generado es 29

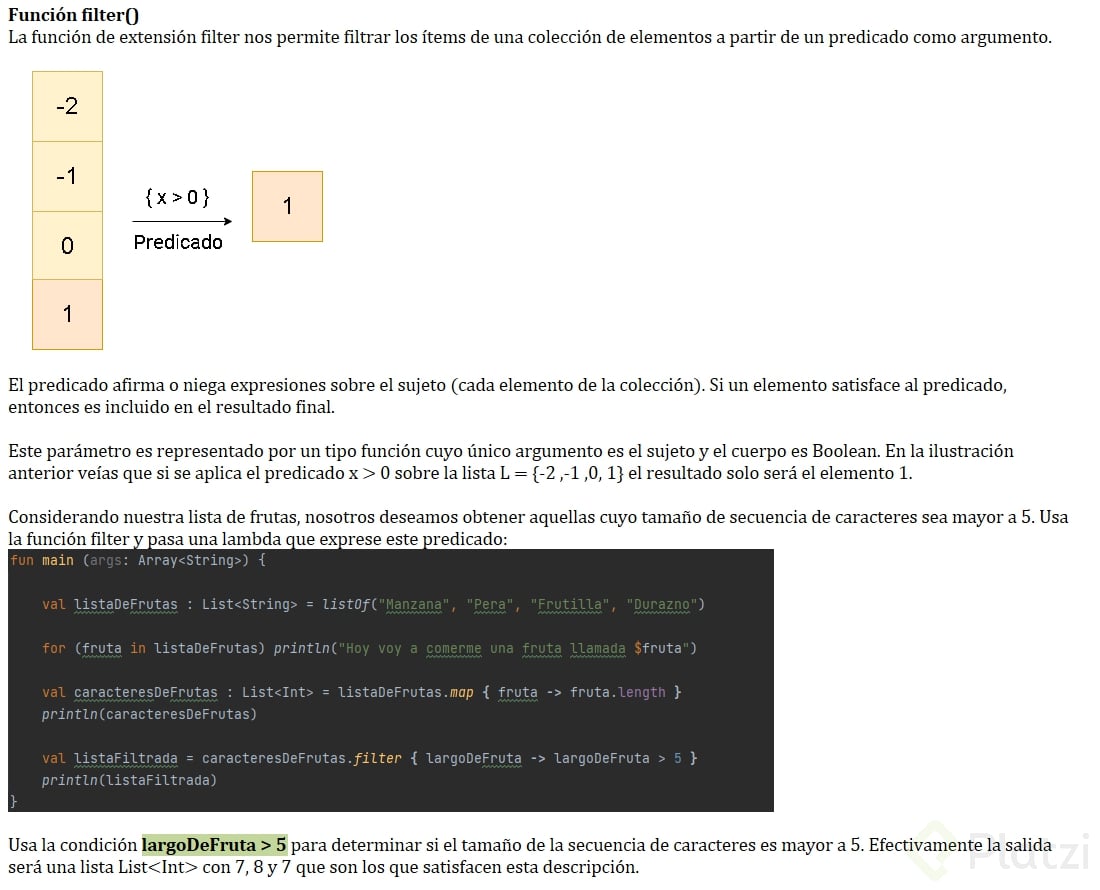
# Ciclos











fun main(args: Array<String>){  
 //for: ciclo for permite ejecutar un codigo para cada elemento de lista que creamos  
 val listaDeFrutas = *listOf*("Manzana"**,** "Pera"**,** "Frambuesa"**,**"Durazno") //list0ftodo lo que agreguemos va a una lista  
 for (fruta in listaDeFrutas) *println*("hoy voy a comerme una fruta llamada $fruta")  
 listaDeFrutas.*forEach***{** fruta**->** *println*("hoy voy a comerme una fruta nueva se llama $fruta") **}** // {}=funcion anonima ejecuta una cantidad n  
 //forEach es lo mismo que: for (fruta in listaDeFrutas)  
  
 //funcion map:  
 val caracteresDeFruta: List<Int> = listaDeFrutas.*map* **{** fruta **->** fruta.length **}** //fruta -> lo convierte a  
 *println*(caracteresDeFruta)  
  
 //funcion filter filtras elementos de una función  
 val listaFiltrada = caracteresDeFruta.*filter* **{** largoDeFrutas **->** largoDeFrutas > **5 }** *println*(listaFiltrada)  
}

hoy voy a comerme una fruta llamada Manzana

hoy voy a comerme una fruta llamada Pera

hoy voy a comerme una fruta llamada Frambuesa

hoy voy a comerme una fruta llamada Durazno

hoy voy a comerme una fruta nueva se llama Manzana

hoy voy a comerme una fruta nueva se llama Pera

hoy voy a comerme una fruta nueva se llama Frambuesa

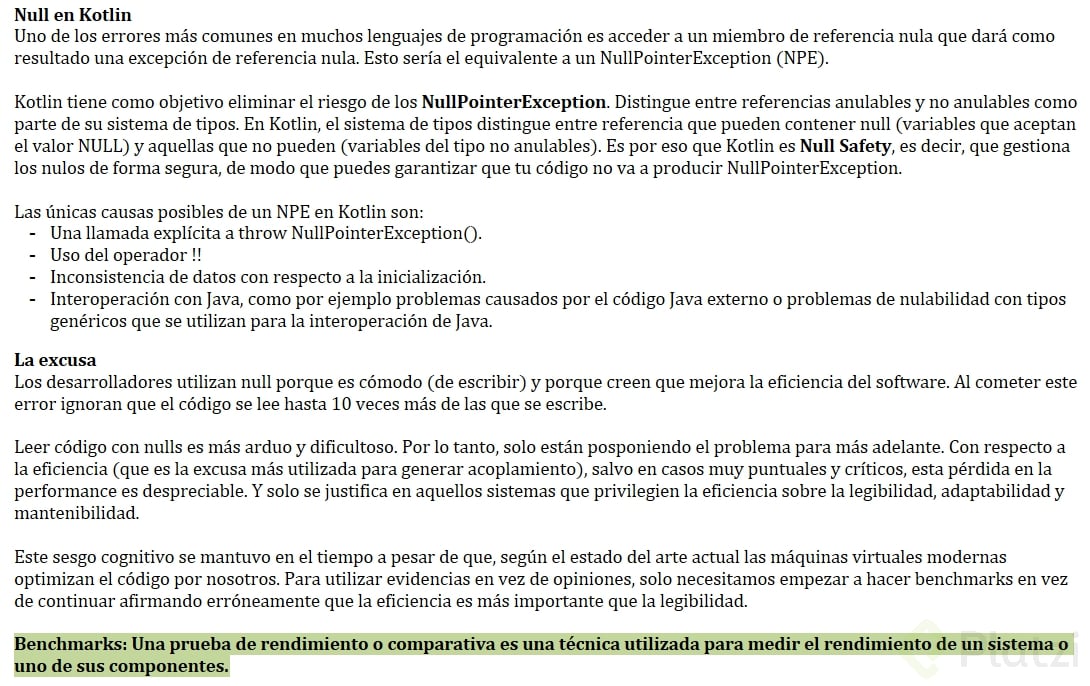
hoy voy a comerme una fruta nueva se llama Durazno

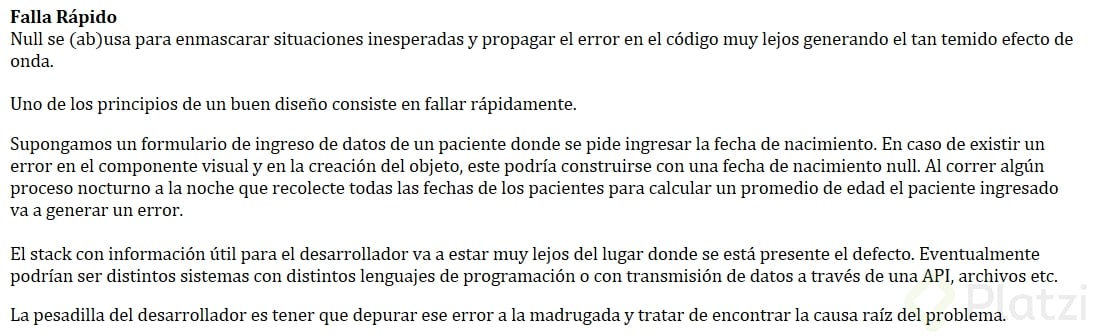
[7, 4, 9, 7]

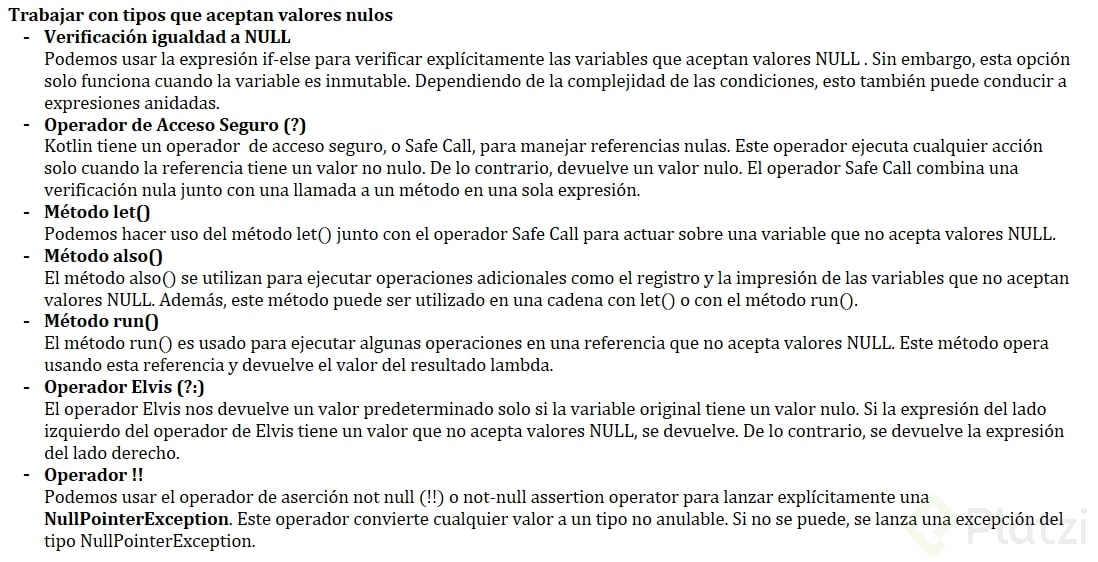
[7, 9, 7]

# Null-Safety en Kotlin



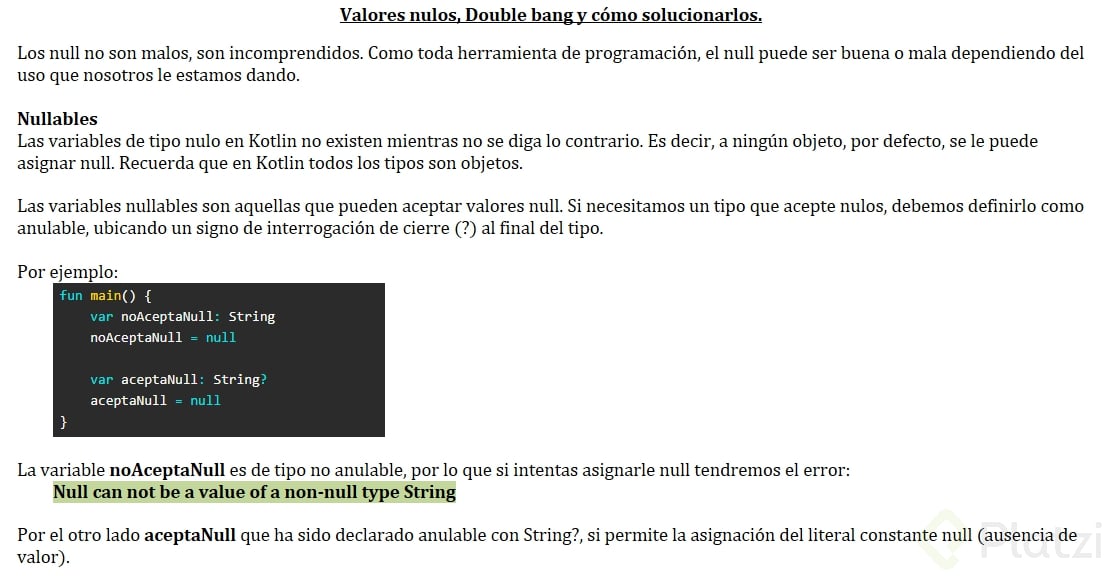


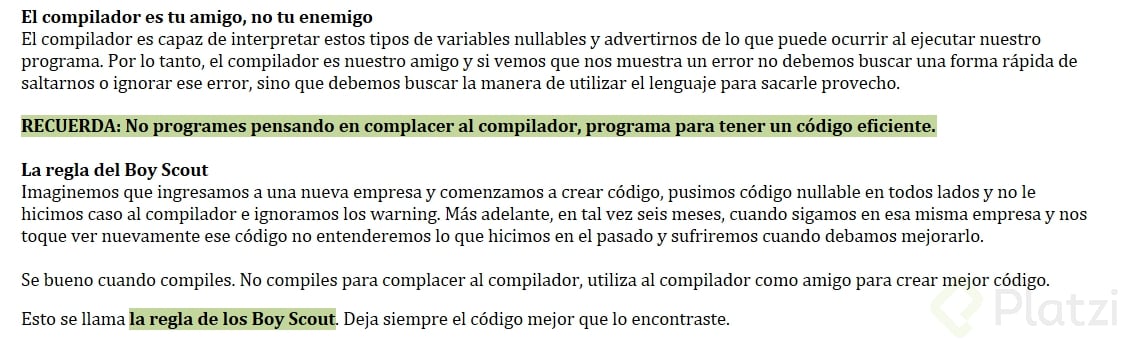


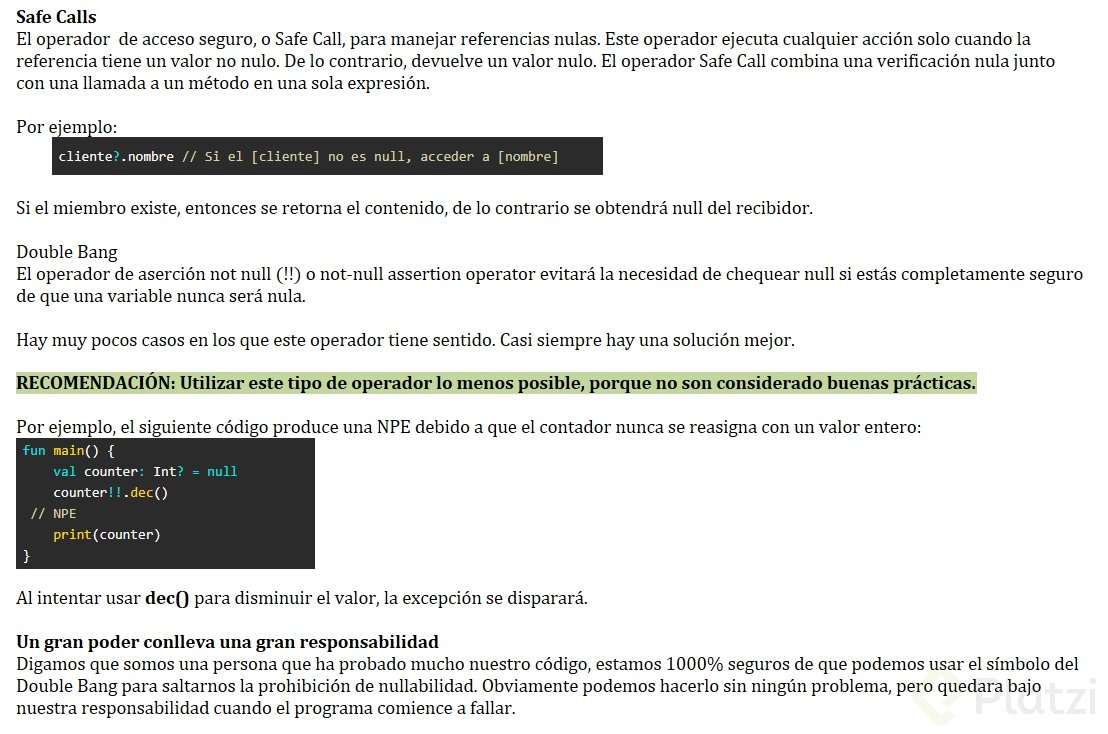


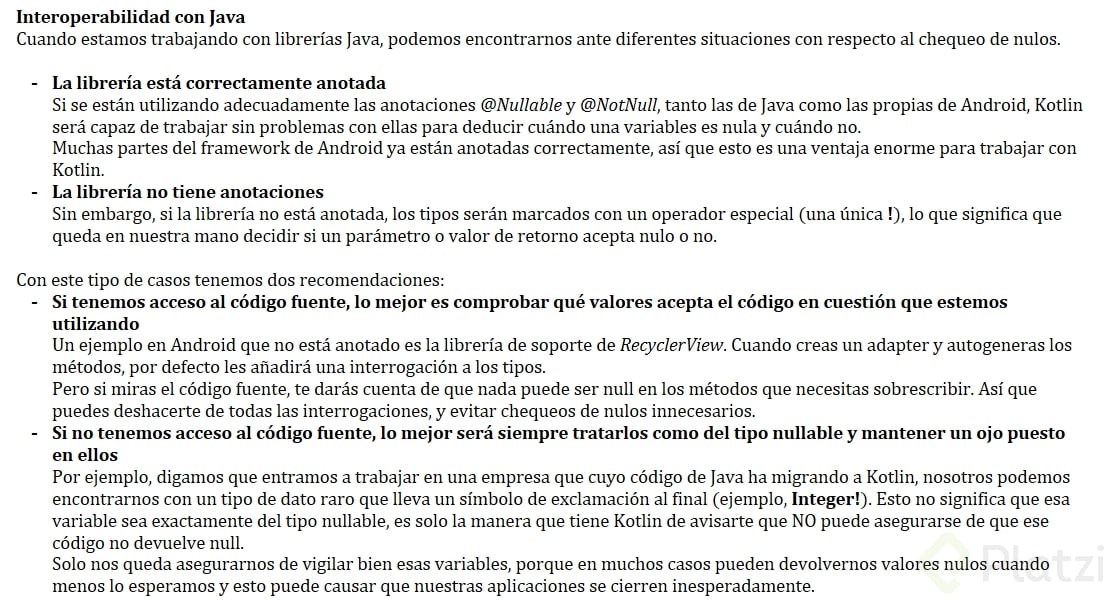
# Valores nulos, Double bang y cómo solucionarlos.

<https://programandoointentandolo.com/2018/02/kotlin-gestion-nulos-null-safety.html>

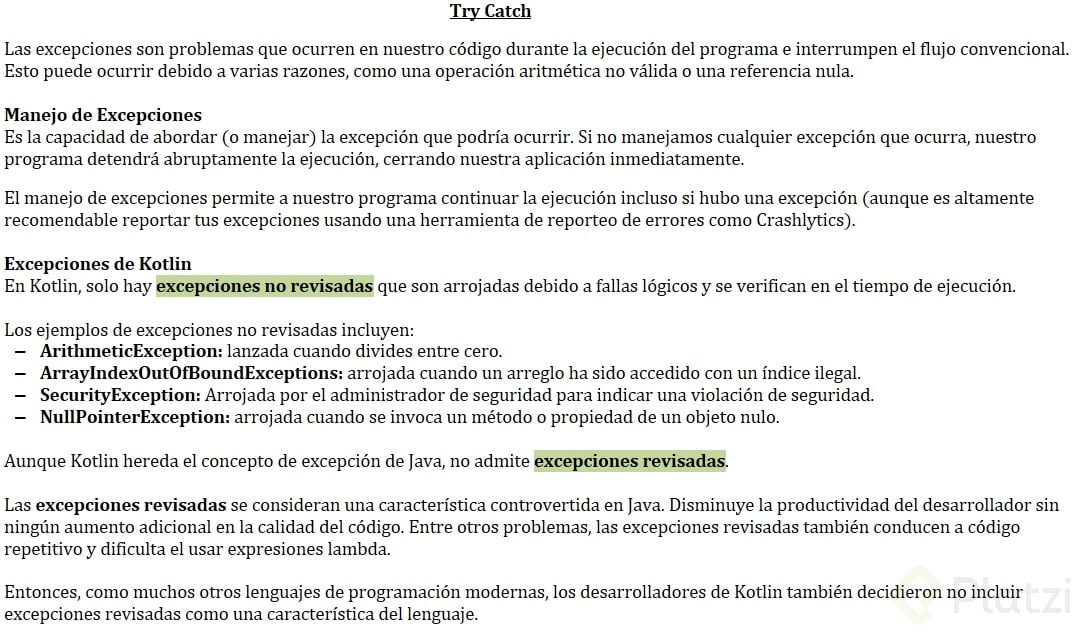


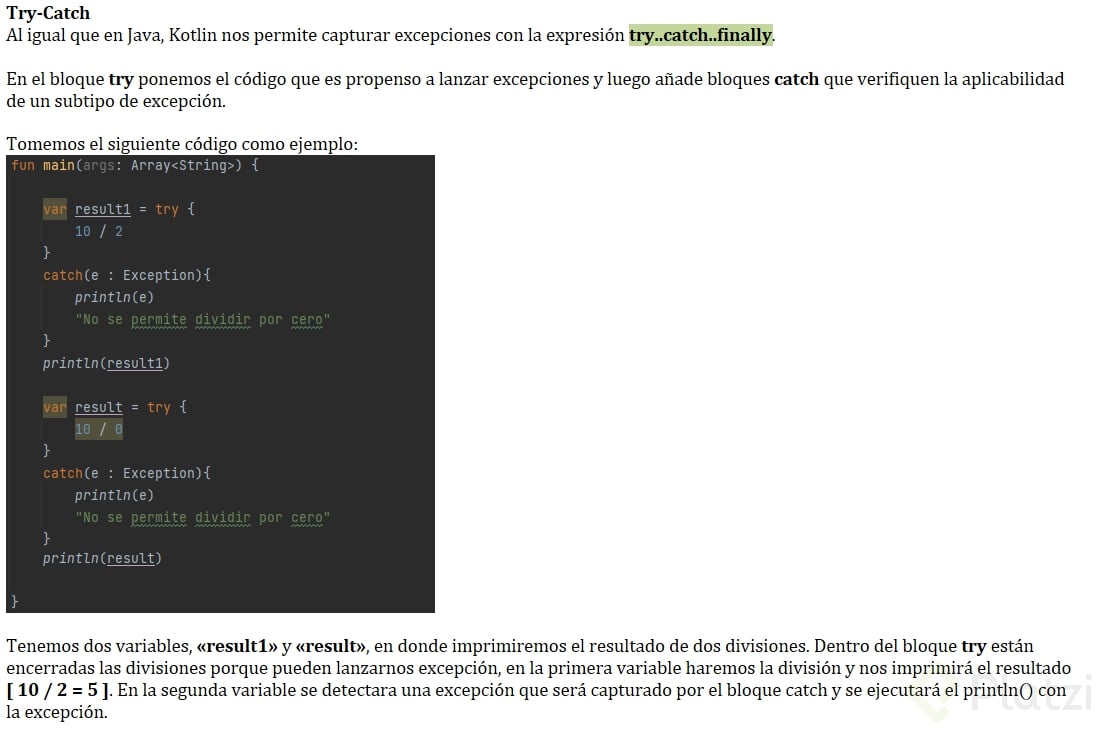


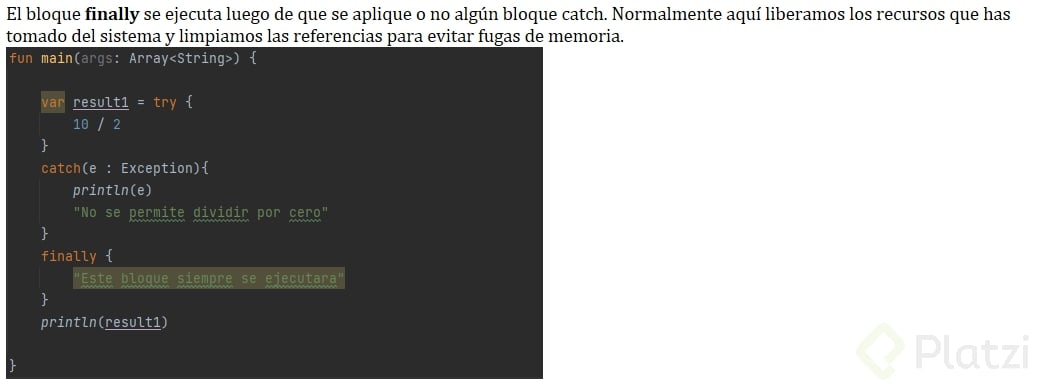




# Try Catch









# Elvis operator

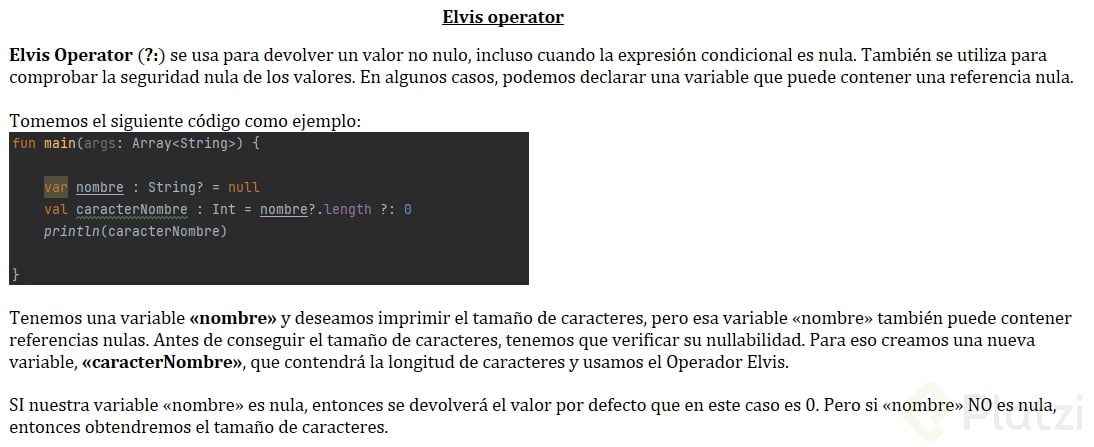
El Elvis operator ?: Es una versión segura de una expresión if. Devuelve el valor a su izquierda si no es nulo. De lo contrario, devuelve el valor a su derecha… por ejemplo:

w?.play ?: -1

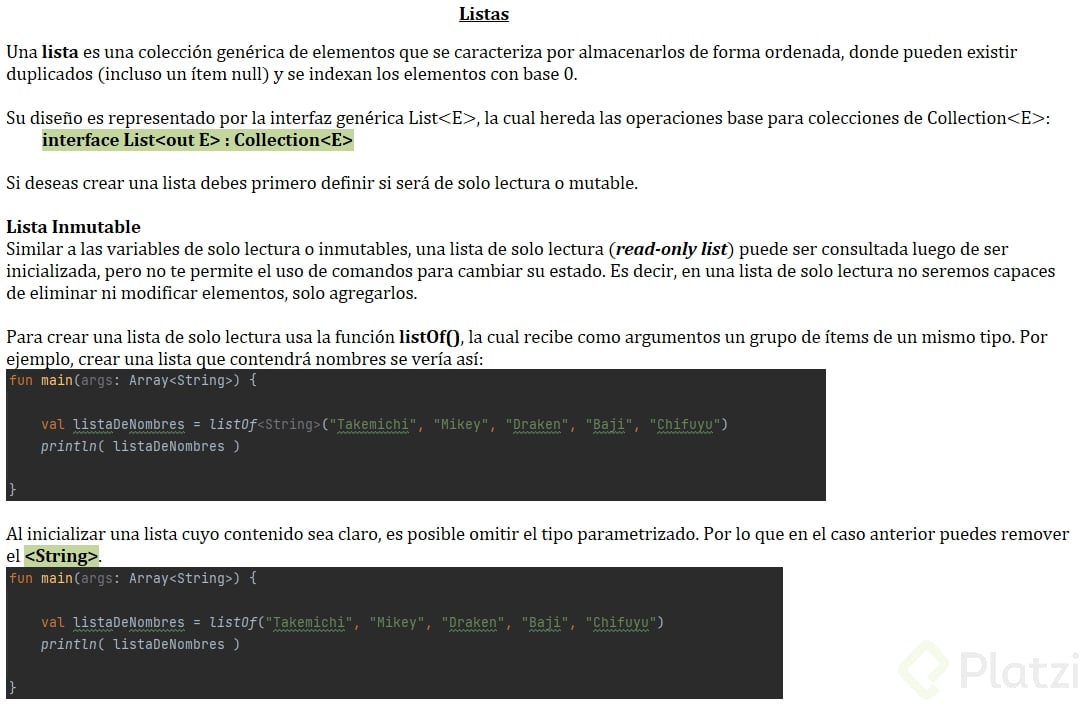
El Elvis operator primero verifica el valor a su izquierda, en este caso w?.play y si ese valor no es null, el Elvis operator lo retorna. En el caso de que el valor de la izquierda sea null, el Elvis operator retornara el valor de la derecha, en este caso -1

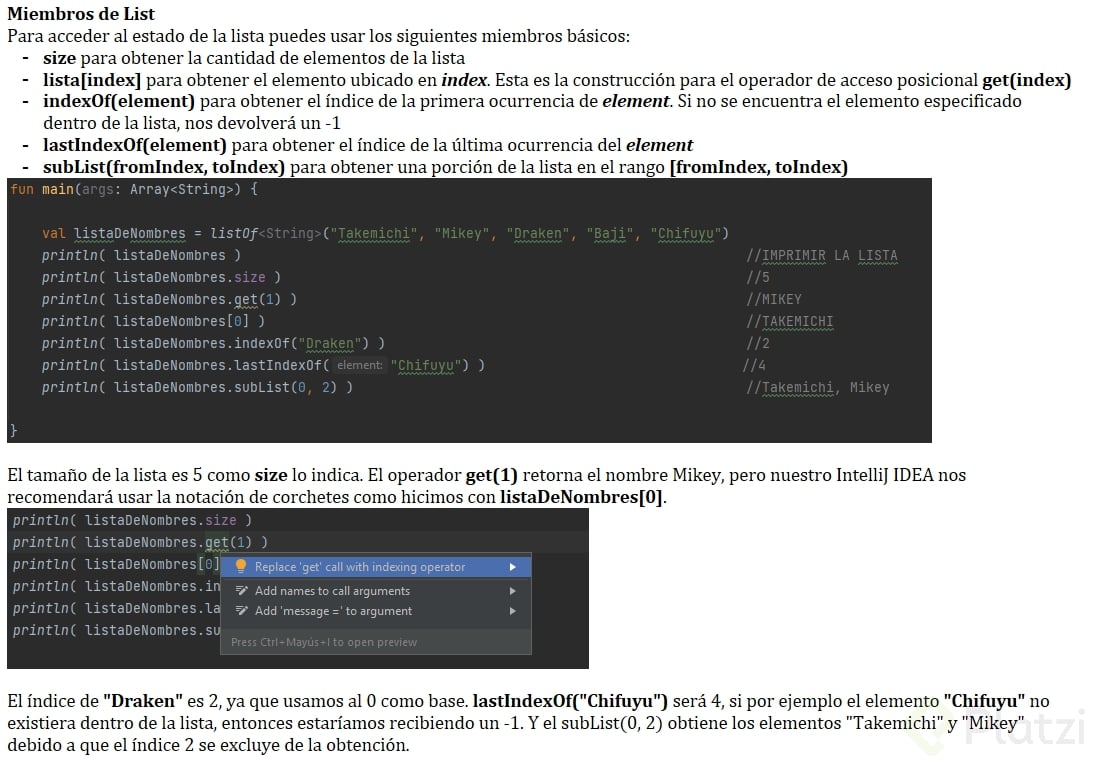
es como decir "si w no es nulo y su propiedad de play no es nula, devuelve el valor de la propiedad de play, de lo contrario, devuelve -1

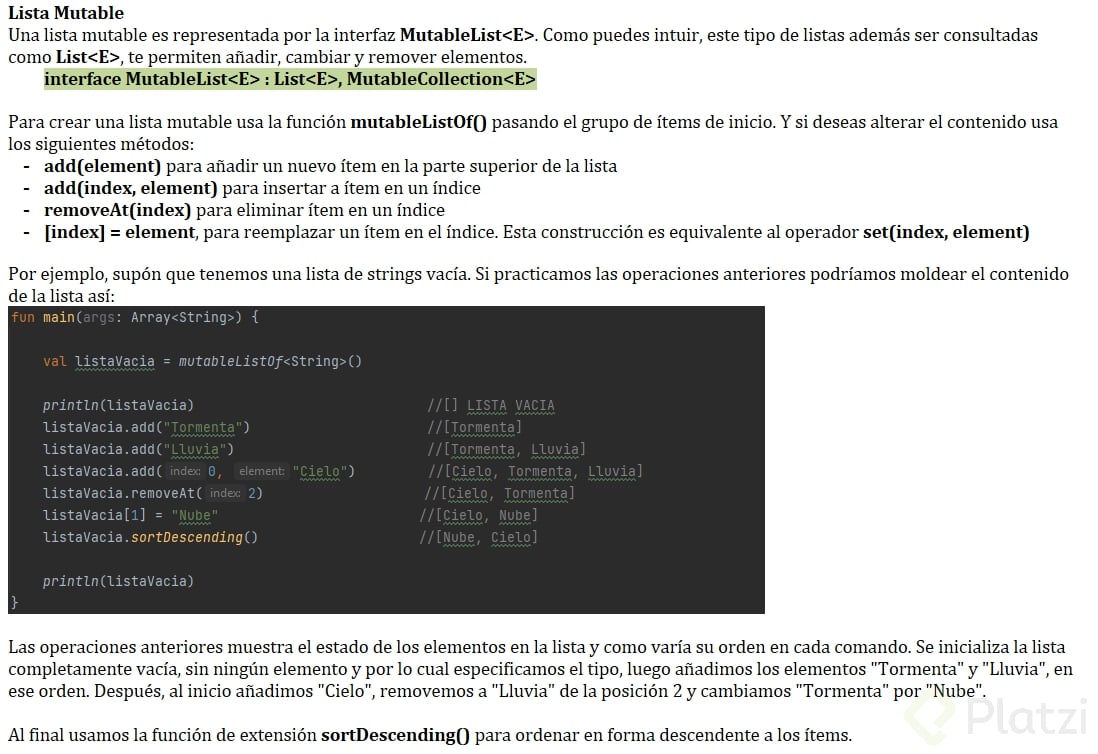
Pueden encontrar mas información sobre este tema en el libro que les recomende anteriormente Head First Kotlin!

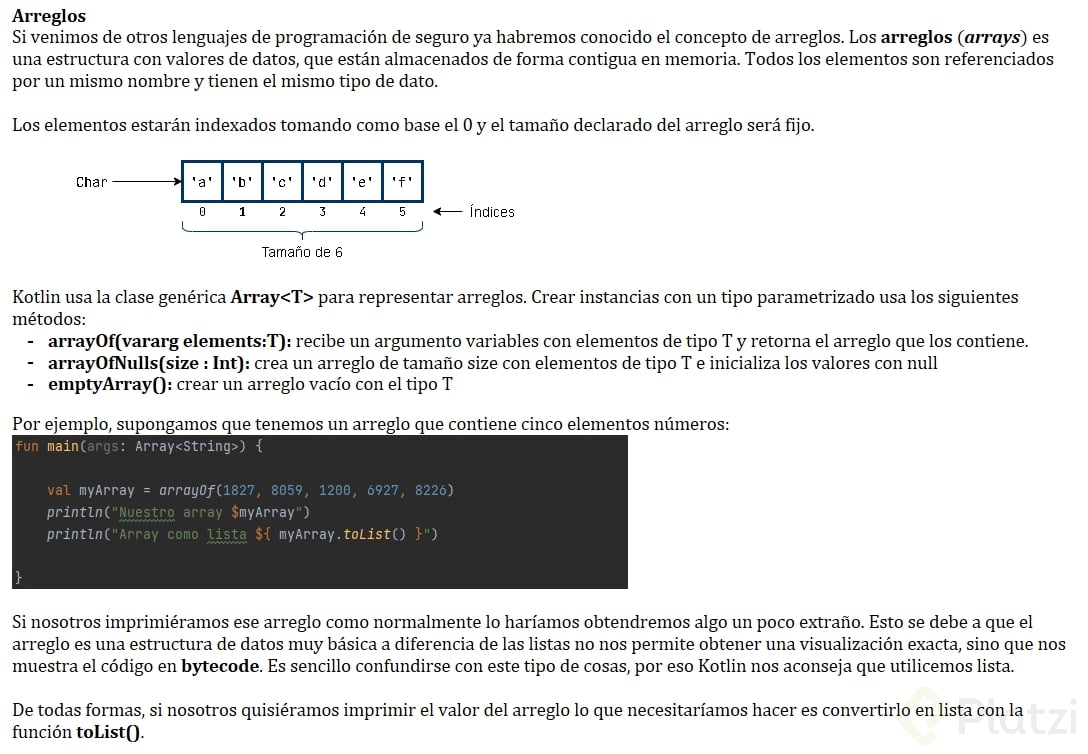


# Listas





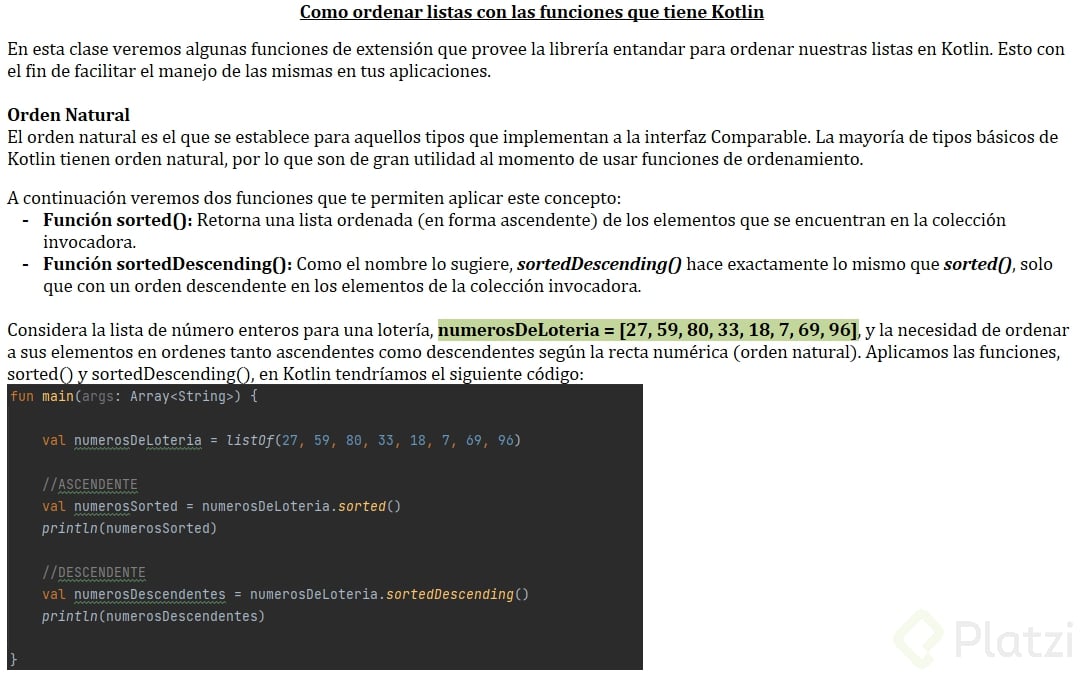


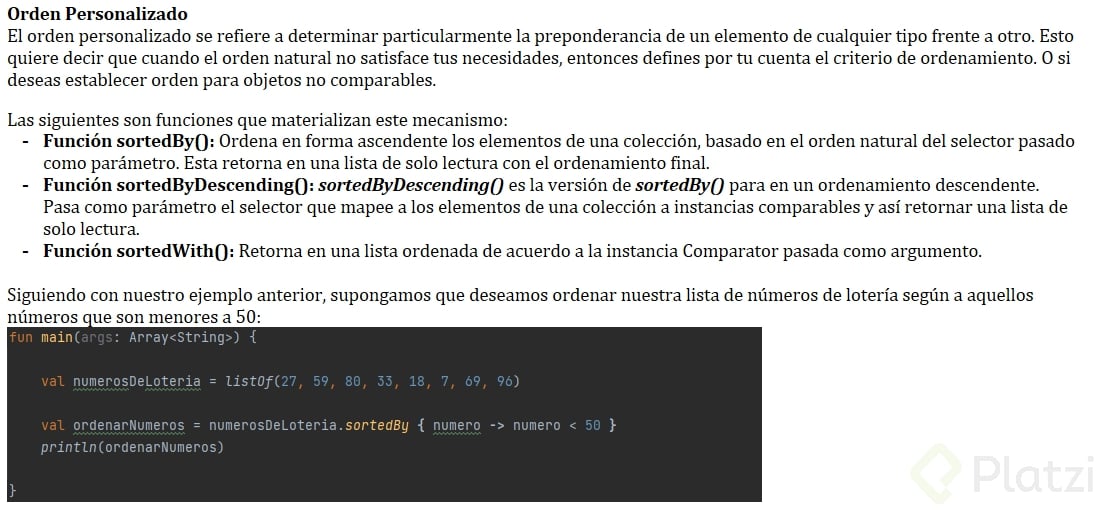


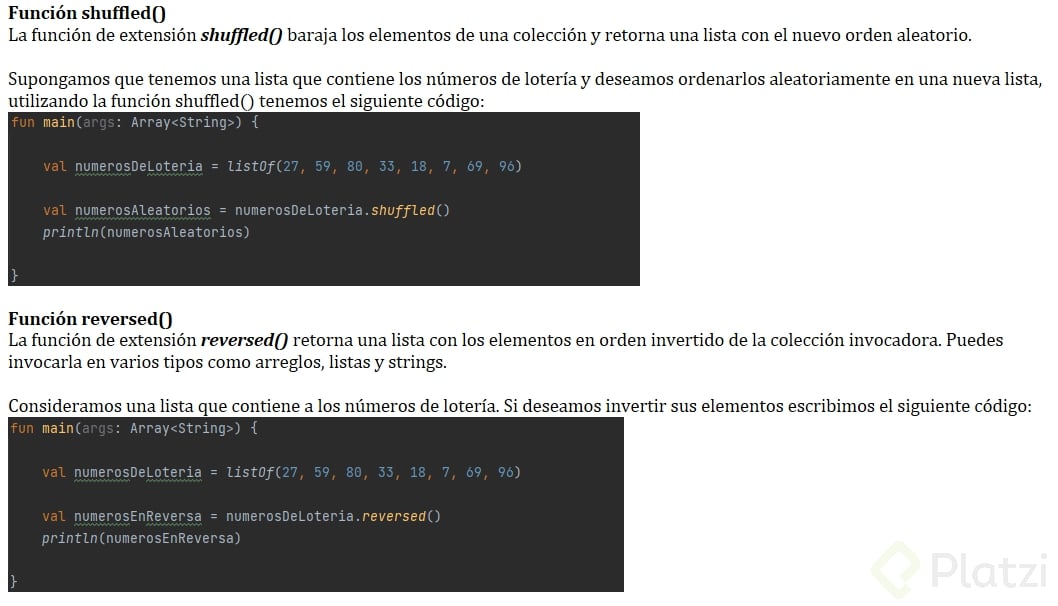
Excelente clase. Recuerden!

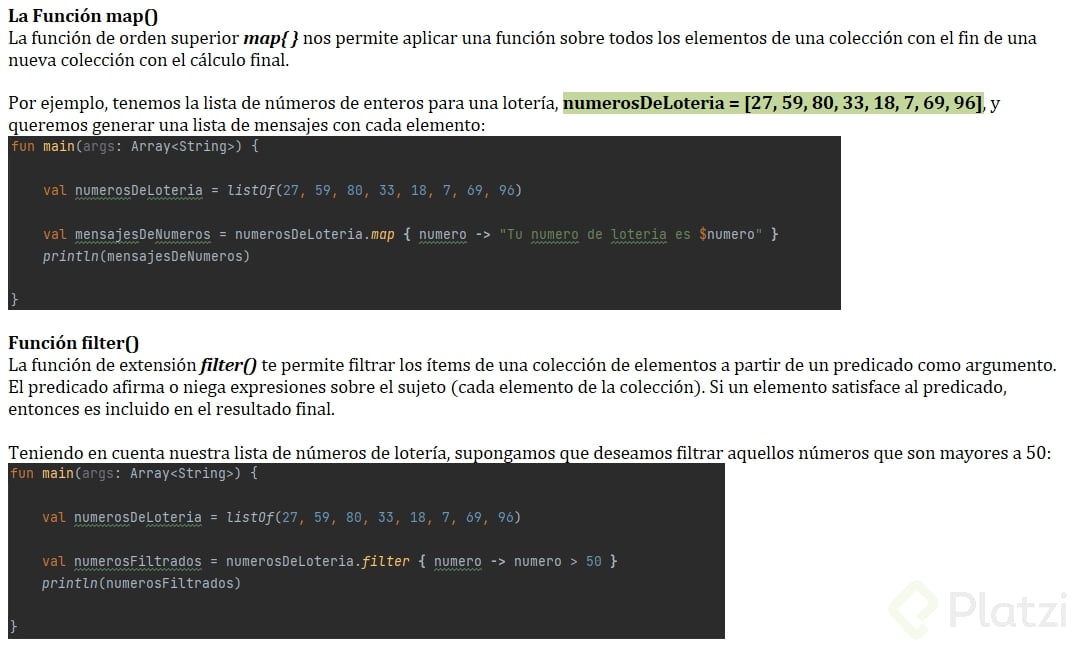
* Las listas son inmutables, si queremos agregar, remover o usar funciones mas avanzadas necesitaremos una mutableList.
* Podemos tener valores duplicados en una lista
* Podemos recorrer todos los elementos de una lista
* La principal diferencia entre una lista y un array es que la lista no puede actualizar ninguna de las referencias que almacena

# Como ordenar listas con las funciones que tiene Kotlin

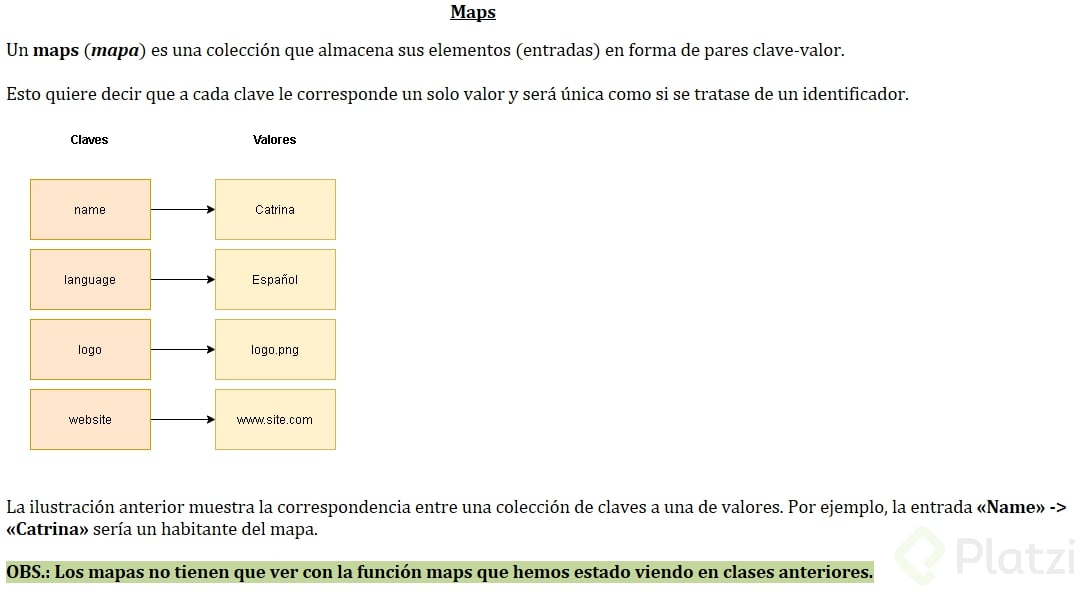


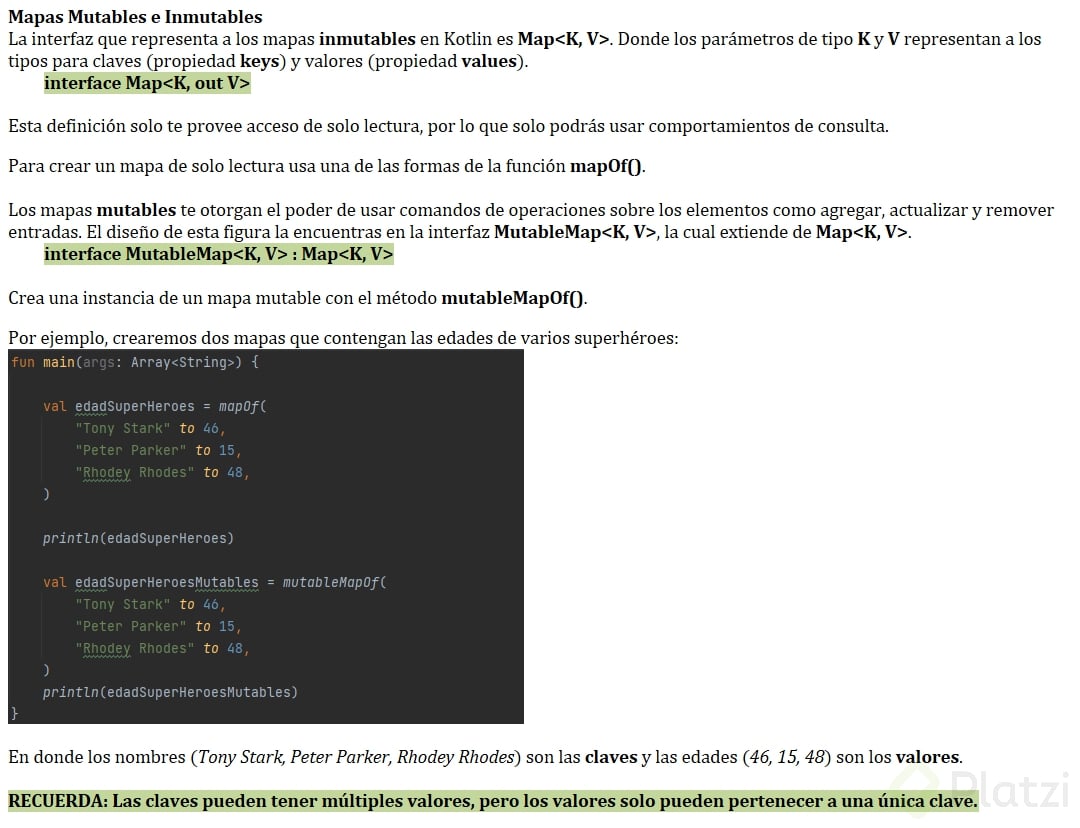






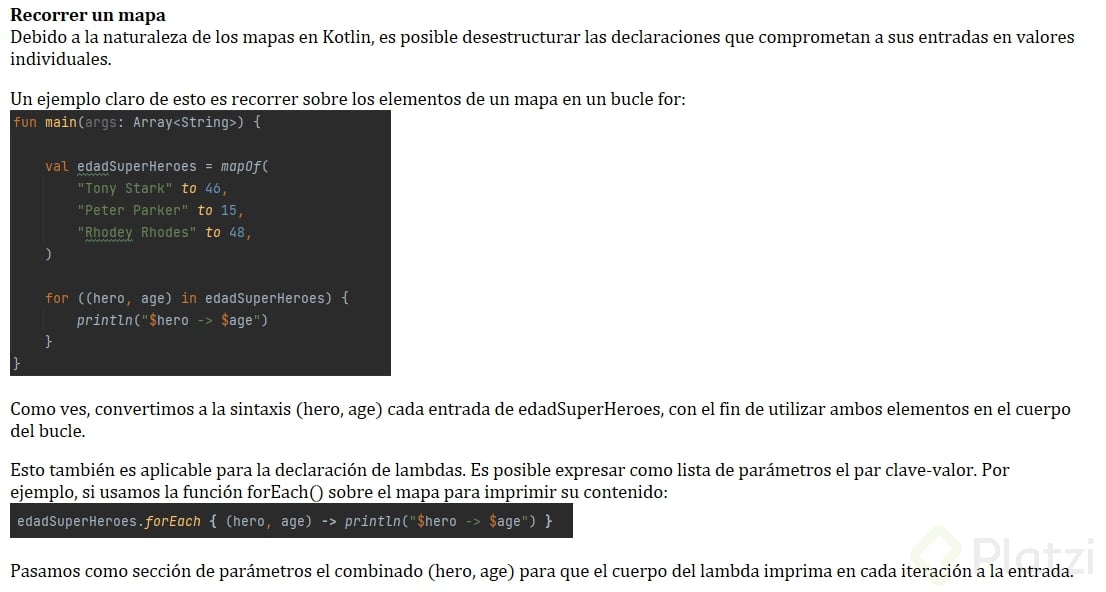
# Maps











**Mapas**  
Los mapas asocian claves con valores. Las claves deben ser únicas, pero los valores asociados no. De este modo, cada valor puede ser usado para identificar de manera única el valor asociado, ya que el mapa asegura que no puedes duplicar claves en la colección. Internamente, Kotlin usa la colección Java Map para implementar los mapas.  
A diferencia de las interfaces List y Set en Kotlin que extienden la interfaz Collection, la interfaz Map no extiende nada. Algunas de las propiedades y funciones disponibles en esta interfaz se muestran a [g1] continuación. Observa como solo se permite hacer consultas, al definir una colección inmutable.

* size: tamaño de la colección.
* isEmpty(): indica si el mapa está vacío.
* containsKey(key: K): indica si el mapa contiene una clave.
* containsValue(value: V): indica si el mapa contiene un valor.
* get(key: K): valor asociado a la llave dada o null si no se encuentra.
* keys: devuelve un Set inmutable con todas las claves en el mapa.
* values: Collection inmutable de todos los valores en el mapa.

mapOf() crea un mapa inmutable compuesto por una lista de pares, donde el primer valor es la clave, y el segundo es el valor. Devuelve un objeto de tipo Map.

val prefijos: Map<Int, String> = mapOf(34 **to** "España", 1 **to** "USA",

233 **to** "Ghana")

**for** ((key, value) **in** prefijos) {

println("$key es el código telefónico de $value")

}

Podemos obtener el valor de una clave usando la función get(). También podemos usar los corchetes como un atajo para get().

**print**(prefijos.get(34)) // España

**print**(prefijos[34]) // España

La interfaz MutableMap no extiende la interfaz MutableCollection; su único padre es la interfaz Map. Este anula las propiedades keys, entries y values de la interfaz padre para poder redefinirlas. Además, incluye algunas funciones extra como:

* put(key: K, value: V) inserta el par clave-valor en el mapa. Devolverá el valor previo enlazado con la clave o null si la clave no existía.  
  r
* emove(key: K) borra la clave y su valor enlazado.
* putAll(from: Map<out K, V>) agrega nuevos pares clave-valor desde otro mapa. Si una clave ya existente será actualizada con el nuevo valor.
* clear() elimina todos los elementos del mapa.

mutableMapOf() permite crear un mapa mutable sin indicar la implementación:

val monedas: MutableMap<String, String> = mutableMapOf("euro" to "España",

"dolar" to "EEUU", "libra" to "UK")

**println**("Paises ${ monedas.values}")

**println**("Monedas ${ monedas.keys}")

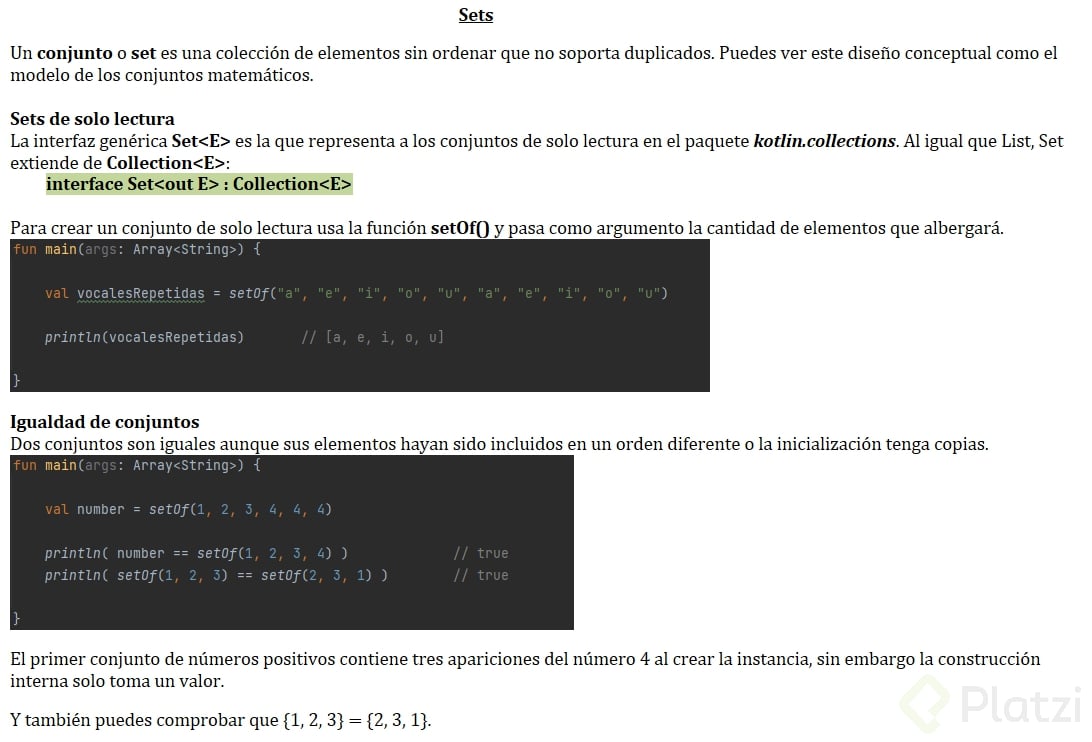
monedas.put("cedi", "Ghana")

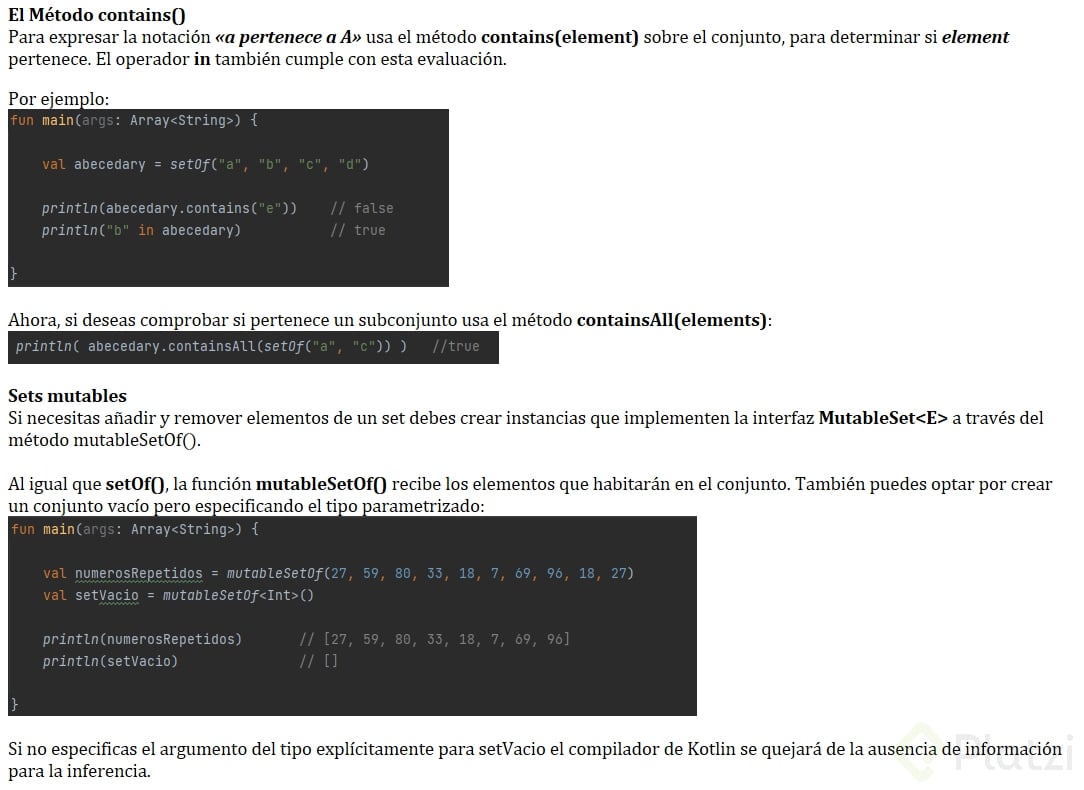
monedas.remove("dolar")

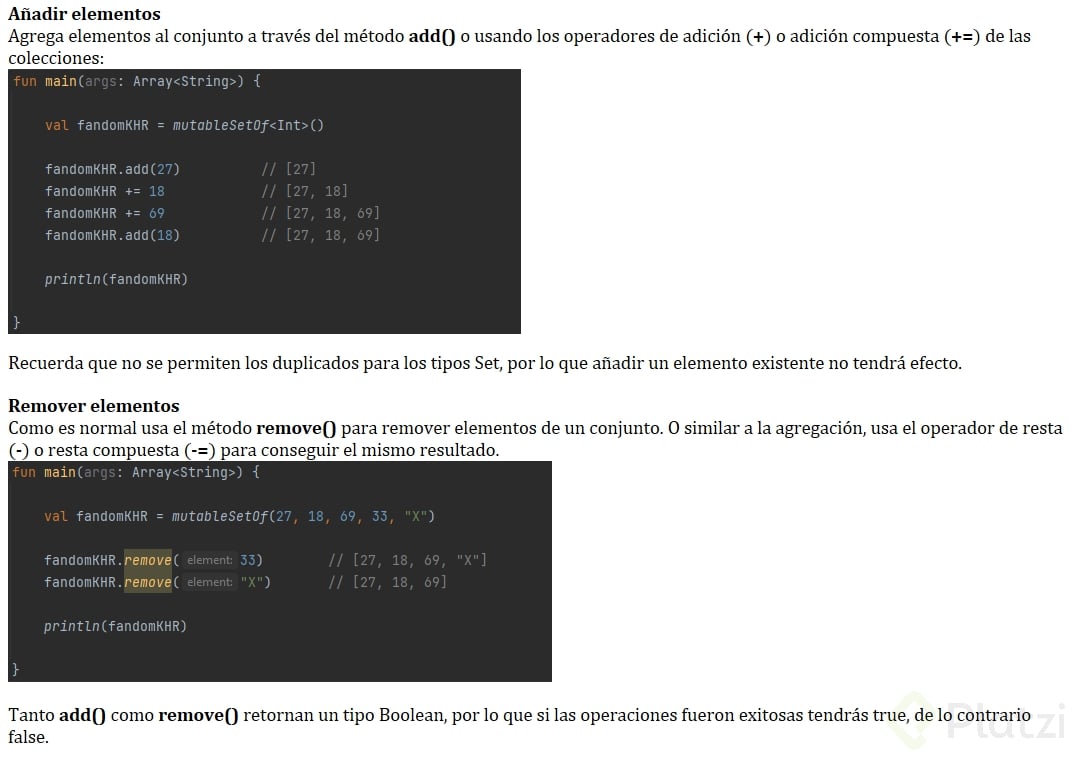
Para indicar implementaciones específicas dispones de: hashMapOf() para crear un mapa de tipo LinkedHashMap., donde puedes consultar el orden en que los elementos fueron insertados, y sortedMapOf() para SortedMap, en el cual todas las entradas se almacenan en un orden de clave ascendente.

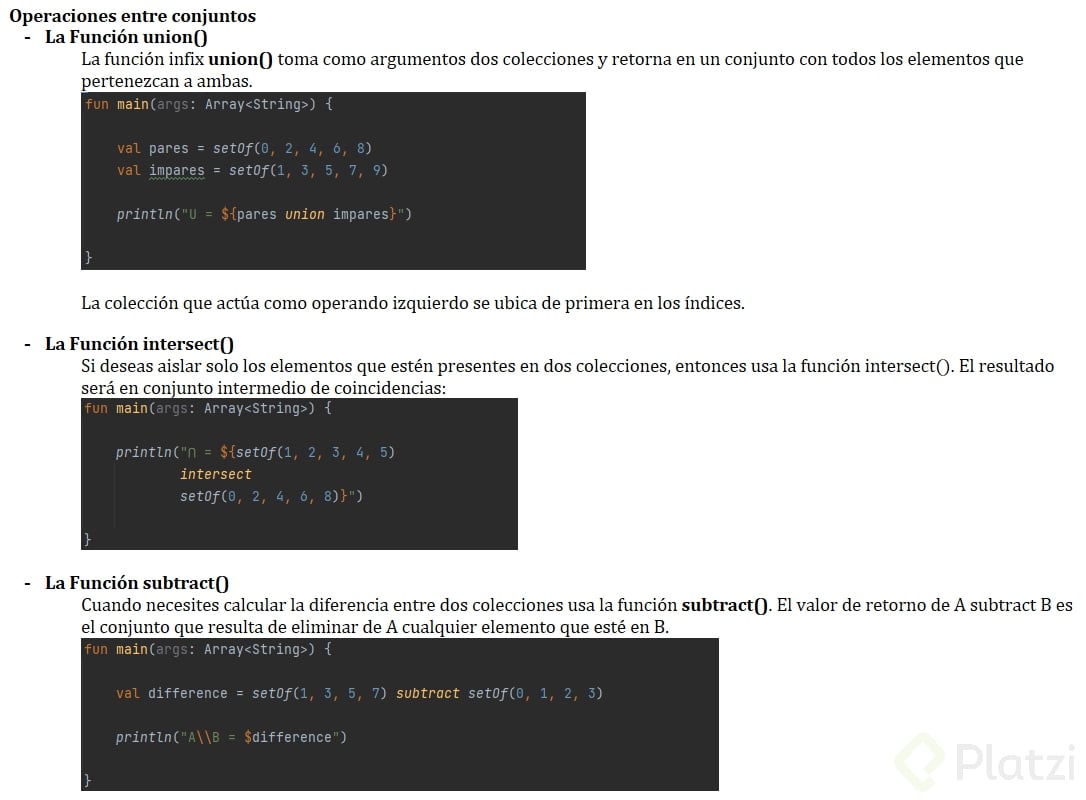
*fuente:*[*https://www.androidcurso.com/index.php/99-kotlin/925-colecciones-en-kotlin-list-set-y-map*](https://www.androidcurso.com/index.php/99-kotlin/925-colecciones-en-kotlin-list-set-y-map)

# Sets









# ¿Qué son las funciones?

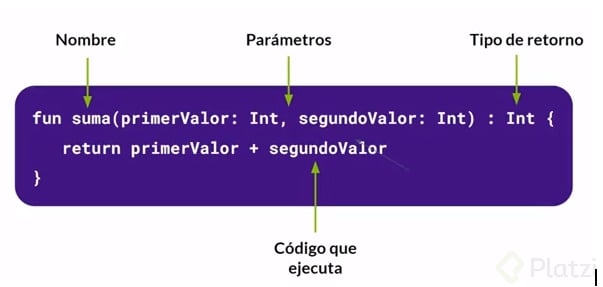
Una función es un código que se ejecuta cada vez que lo llamamos.

**Sintaxis de una función**

Las funciones más básicas se componen de 4 partes.

* Palabra reservada fun.
* Nombre de la función.
* Parámetros: Son las variables que le damos a la función para que las use en el código que ejecuta internamente.
* Tipo de retorno: Puede tener o no un valor de retorno.
* Código: Son las instrucciones que se van a ejecutar al llamar a la función.

Ejemplo de función:



Cuando queremos devolver algo de nuestra función usamos la keyword **return**.  
Caso contrario cuando no queremos devolver nada de nuestra función, Kotlin regresaría **Unit**.

Ejemplo de función que no devuelve nada, no tiene ningún tipo de retorno.



# Funciones y funciones de extensión

**Funciones de extensión en Kotlin**  
Las funciones de extensión (o extension functions en inglés) son funciones que, como su propio nombre indica, nos ayudan a extender la funcionalidad de clases sin necesidad de tocar su código. Ahora vamos a ver cómo se definen estas funciones, y algunos ejemplos que a mí personalmente me resultan muy útiles.

**¿Cómo se define una función de extensión?**  
Tan solo hay que escribir una función como lo harías normalmente, y ponerle delante el nombre de la clase separado por un punto.

Ejemplo muy sencillo: queremos hacer que una vista tenga la función visible(), que la hace visible. Escribiríamos algo como esto:

**fun** View.**visible**() {

**this**.visibility = View.VISIBLE

}

El **this** lo he puesto para que veas que podemos usar las funciones y propiedades de esa clase como si estuviéramos dentro de la propia clase, pero lo puedes omitir:

**fun** View.**visible**() {

visibility = View.VISIBLE

}

**Algunos ejemplos interesantes**  
Hay un par de ejemplos que me gusta poner, porque resumen muy bien la potencialidad de esto.

El primero es cuando estás inflando una vista dentro de un adapter. Normalmente utilizarías algo así:

**override** **fun** **onCreateViewHolder**(parent: **ViewGroup**, viewType: **Int**): ViewHolder {

**val** v = LayoutInflater.from(parent.context).inflate(R.layout.view\_item, parent, **false**)

**return** ViewHolder(v)

}

La línea que infla la vista y usa el parent es demasiado compleja, y el 99% de las veces suele ser igual en cualquier adapter. ¿Por qué no hacer que los ViewGroup puedan inflar vistas?

**fun** ViewGroup.**inflate**(layoutRes: **Int**): View {

**return** LayoutInflater.from(context).inflate(layoutRes, **this**, **false**)

}

Ahora ya puedes utilizarlo en el código de arriba:

**override** **fun** **onCreateViewHolder**(parent: **ViewGroup**, viewType: **Int**): ViewHolder {

**val** v = parent.inflate(R.layout.view\_item)

**return** ViewHolder(v)

}

Un ejemplo muy parecido se puede hacer con las imágenes. Si utilizas por ejemplo la librería de Picasso, necesitas andar haciendo el típico ritual:

**Picasso**.with(imageView.context).load(url).into(imageView)

¿Qué te parecería poder decirle a ImageView que cargue una url directamente?

**fun** ImageView.loadUrl(url: String) {

Picasso.**with**(context).load(url).into(this)

}

imageView.loadUrl(url)

**Propiedades de extensión**  
Igual que puedes hacer funciones de extensión, lo mismo puedes hacer con properties. Lo único que no podrán guardar un estado propio, sino valerse de las funciones ya existentes para modificar el estado:

**val** ViewGroup.children: List

get**()** = (0..childCount -1).map { getChildAt(it) }

Esta property recupera los hijos de un ViewGroup

Ahora podrías iterar sobre ellos directamente:

parent.children.forEach { it.visible() }  
Nota: it es una palabra reservada que se utiliza para acceder al valor de entrada de la función, cuando solo hay uno. Como ya hemos visto en otros artículos, se pueden nombrar esos valores de entrada, y asignar más cuando hay más de uno.

**Conclusión**  
Con las funciones y las propiedades de extensión puedes extender cualquier librería a la que no tengas acceso y luego utilizar esas funciones y propiedades como si fueran propias de la clase. Lo único que verás es un import extra en el archivo en el que se use.

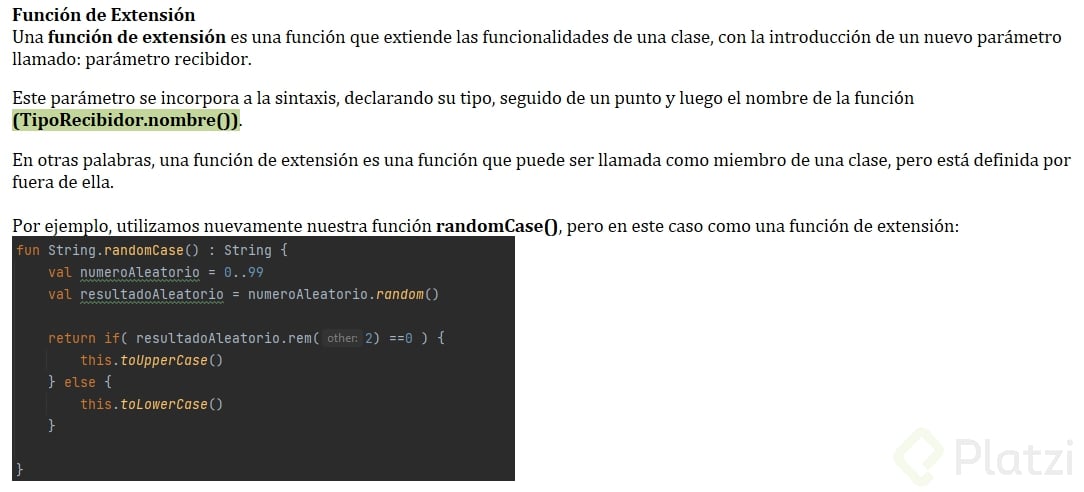
Si de verdad vas en serio con Kotlin y, como yo, piensas que es el lenguaje del futuro en Android, te recomiendo que le eches un vistazo al training gratuito, donde te contaré todo lo que necesitas para aprender a crear tus Apps Android en Kotlin desde cero.

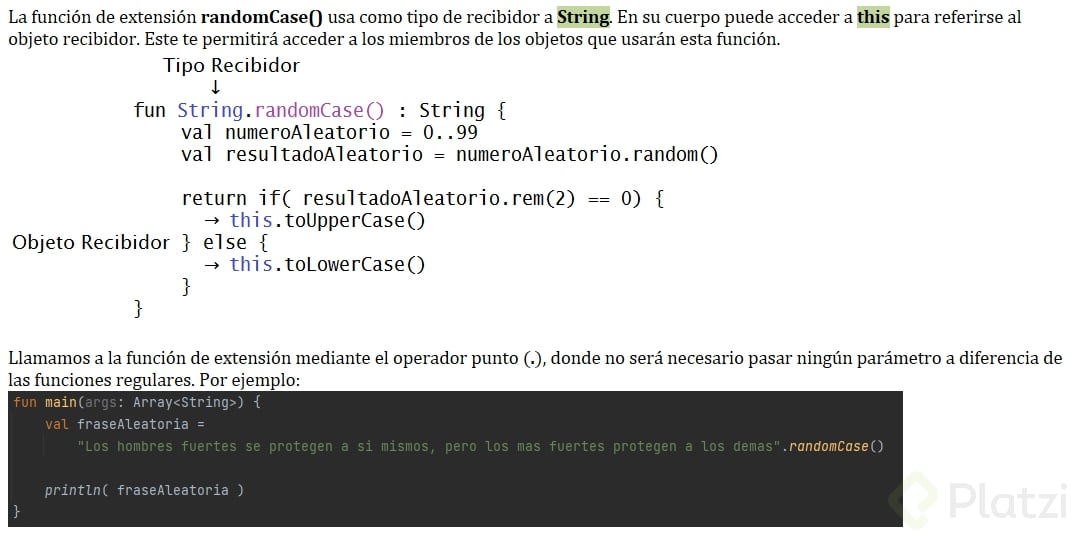
*fuente:*[*https://devexperto.com/funciones-extension-kotlin-android/*](https://devexperto.com/funciones-extension-kotlin-android/)

**toUpperCase()** y **toLowerCase()** han sido marcadas como obsoletas. En su lugar recomienda usar: **uppercase()** y **lowercase()**









# Tipos de parámetros en las funciones

