



# UD2.2 – Fundamentos de Kotlin

**2º CFGS**  
**Desarrollo de Aplicaciones Multiplataforma**  
**2023-24**

# 1.- Control de flujo

Instrucciones que permiten alterar el flujo normal de las instrucciones del programa.

- if
- if – else
- if – else if – else
- when
- for
- while
- do – while
- repeat

# 1.- Control de flujo

Guía de estilo para las llaves { }:

- Las llaves **no son necesarias** para ramas de **when** ni expresiones de **if** que **no tengan más de una rama de else** y que **quepan en una sola línea**.
- Las llaves **son necesarias** para cualquier rama de **if**, **for** o **when, do**, y para sentencias y expresiones de **while**, incluso cuando el cuerpo está vacío o contiene una sola instrucción.

## 2.- if, if-else y if-else if-else

Si se puede escribir en una sola línea no se ponen las llaves y es preferible escribirlo así.

```
if (number == 0) {  
    println("El número es cero")  
}
```

```
if (number == 0) println("El número es cero")
```

```
if (number1 > number2) {  
    higherNumber = number1  
} else {  
    higherNumber = number2  
}
```

```
if (number1 > number2) higherNumber = number1 else higherNumber = number2
```

```
if (number1 > number2) {  
    println("El mayor es: $number1")  
} else if (number1 < number2) {  
    println("El mayor es: $number2")  
} else {  
    println("Los dos números son: $number1")  
}
```

### 3.- if como una expresión

La instrucción **if** se puede utilizar como una expresión, por ejemplo para asignar un valor a una variable como en el ejemplo ya visto:

```
if (number1 > number2) higherNumber = number1 else higherNumber = number2
```

En este caso se puede escribirla expresión de la siguiente forma, siendo obligatorio incluir la rama del **else**.

```
higherNumber = if (number1 > number2) number1 else number2
```

Si se usa como una expresión **la rama de else es obligatoria**.

### 3.- if como una expresión

Al usar la instrucción **if** como expresión también se pueden utilizar las llaves para realizar varias instrucciones, en este caso la última instrucción de las llaves debe ser el valor a asignar.

Si se usa como expresión y además, se usan bloques de código la última expresión debe ser el valor a asignar sin necesidad de igualar a la variable.

```
var highNumber = if (number1 > number2) {  
    // instrucciones  
    number1  
} else {  
    // instrucciones  
    number2  
}  
println("El mayor es $highNumber")
```

## 4.- when

Condición con múltiples ramas (similar a switch de Java):

```
when (number) {  
  1 -> println("La variable es 1")  
  2 -> println("La variable es 2")  
  else -> println ("La variable no es ni 1 ni 2")  
}
```

Si hay varias instrucciones en alguna rama, se debe poner las llaves a todas las ramas:

```
when (number) {  
  1 -> {  
    // instrucciones  
  }  
  2 -> {  
    // instrucciones  
  }  
}
```

## 4.- when

Con **when** se pueden agrupar valores:

```
when (position) {  
    1, 2, 3 -> println("En el podio")  
    else -> println("Sigue intentándolo")  
}
```

También se pueden usar expresiones para evaluar:

```
var number:Int = 2  
var position:String = "5"  
when (number) {  
    parseInt(position) -> println("coincide el valor numérico")  
    else -> println("no coincide el valor numérico")  
}
```



## 4.- when

Se puede usar **in** o **!in** para usar **rangos** y **colecciones** para evaluar:

```
when (position) {  
    in 1 ≤ .. ≤ 10 -> println("En el top 10")  
    !in 11 ≤ .. ≤ 15 -> println("En la mitad")  
    else -> println("Muy atrás")  
}
```

```
val numbers = listOf(1, 2, 3, 4, 5)  
when (position) {  
    in numbers -> println("Top 5")  
    else -> println("Sigue intentándolo")  
}
```

Se puede evaluar si la variable es de un tipo o no con **is** o **!is**:

```
var variable:Any = "Rick"  
when (variable) {  
    is Int -> println("Es un número")  
    !is String -> print("No es una cadena")  
    else -> println("Es de otro tipo")  
}
```

## 4.- when

Hay casos en los que **la rama else es obligatoria**, por ejemplo si el resultado de la expresión a evaluar es un **booleano**:

```
// No compila ya que no se evalúan todas las posibilidades
when (number1 > number2) {
    true -> println("$number1 es mayor que $number2")
}
```

```
// compila porque se evalúan todas las posibilidades
when (number1 > number2) {
    true -> println("$number1 es mayor que $number2")
    else -> println("$number1 no es mayor que $number2")
}
```

```
// compila porque se evalúan todas las posibilidades
when (number1 > number2) {
    true -> println("$number1 es mayor que $number2")
    else -> println("$number1 no es mayor que $number2")
}
```

Si no se indica ninguna variable para evaluar when servirá para sustituir bloques if-else:

```
when {
    number1 > number2 -> print("El mayor es: $number1")
    number1 < number2 -> print("El mayor es: $number2")
    else -> print("Son iguales")
}
```

## 4.- when

Se puede usar como expresión, en ese caso la rama else es obligatoria a menos que el compilador pueda comprobar que se están cubriendo todas las opciones:

```
// se necesita el else para tener todas las posibilidades  
val result: String = when (number) {  
    1 -> "La variable es 1"  
    2 -> "La variable es 2"  
    else -> "La variable no es ni 1 ni 2"  
}
```

```
// no se necesita el else porque ya están todas las posibilidades  
var result: String = when (number1 > number2) {  
    true -> "$number1 es mayor que $number2"  
    false -> "$number1 no es mayor que $number2"  
}
```

## 6.- for

Los bucles for son un poco diferentes en Kotlin debido a que utilizan rangos, progresiones y colecciones para las iteraciones.

- **Rangos:** serie ascendente/descendente de Int, Long o Char.
- **Progresiones:** como un rango pero con un parámetro que indica el paso entre los elementos de la progresión.
- **Colecciones:** conjuntos de elementos como arrays.

## 5.- rangos y progresiones

## Existen diferentes maneras de crear rangos y progresiones.

En la imagen de la izquierda se muestra cómo se crean, pero por defecto IntelliJ IDEA tiene activadas **las pistas en el código** (hints).

Una vez creado el rango se verá como en la imagen de la derecha mostrando más información.

Todos los rangos se pueden usar como progresiones con paso 1 por defecto o se pueden convertir en progresiones indicando el paso explícitamente como en el último ejemplo.

```
0.rangeTo(10)           // rango de 0 a 10 incluido
0..10                   // rango de 0 a 10 incluido

0.rangeUntil(10)        // rango de 0 a 9 incluido
0 until 10              // rango de 0 a 9 incluido
0..< 10                 // rango de 0 a 9 incluido

'a' .. 'z'              // rango de a hasta z incluida

10.downTo(0)            // Progresión de 10 a 0 incluido de 1 en 1
10 downTo 0            // Progresión de 10 a 0 incluido de 1 en 1

0..10 step 2           // Progresión de 0 a 10 incluido de 2 en 2
```

```
0.rangeTo( other: 10)
0 ≤ .. ≤ 10

0.rangeUntil( other: 10)
0 ≤ until < 10
0 ≤ .. < 10

'a' ≤ .. ≤ 'z'

10.downTo( to: 0)
10 ≥ downTo ≥ 0

0 ≤ .. ≤ 10 step 2
```

## 6.- for

Los bucles for utilizan **rangos**, **progresiones** y **colecciones** y para recorrerlos se indica con la palabra **in**.

```
// si el cuerpo ocupa una sola línea mejor no poner bloque
for (i in 1 ≤ .. ≤ 3) print("$i ") // 1, 2, 3
```

```
// el cuerpo del for puede ser un bloque
for (i in 0 ≤ until < 5) {
    // 0, 1, 2, 3, 4
}
```

```
for (i in 10 ≥ downTo ≥ 0) {
    // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
}
```

```
for (i in 1 ≤ .. ≤ 14 step 3) {
    // 1, 4, 7, 10, 13
}
```


```
for (i in 10 ≤ until < 25 step 2) {
    // 10, 12, 14, 16, 18, 20, 22, 24
}
```

```
for (i in 10 ≥ downTo ≥ 0 step 2) {
    // 10, 8, 6, 4, 2, 0
}
```

## 6.- for

También funciona con rangos de caracteres

```
for (letter in 'c' ≤ .. ≤ 'j') {  
  // c d e f g h i j  
}  
  
for (letter in 'A' ≤ .. ≤ 'Z') {  
  // A B C ... X Y Z  
}  
  
//for (letter in '\u0021'..'\'u007E') {  
for (letter in '!' ≤ .. ≤ '~') {  
  // De UTF16 0021 a UTF16 007E  
}  
  
for (letter in 'c' ≤ .. ≤ 'j' step 3) {  
  // c f i  
}  
  
for (letter in 'z' ≥ downTo ≥ 'm' step 2) {  
  // < x v t r p n >  
}
```



! " # \$ % & ' ( ) \* + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = >  
? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \  
] ^ \_ ` a b c d e f g h i j k l m n o p q r s t u v w x y z  
{ | } ~

## 6.- for

La instrucción for permite recorrer objetos que provean un **iterator** como strings, listas, arrays o colecciones.

Se puede acceder a la posición de dos maneras.

```
for (letter: Char in "Rick Sanchez") {  
  
}  
  
var letters = listOf('a', 'b', 'c', 'd')  
for (letter: Char in letters) {  
  
}  
  
for (letter in letters) { //se deduce el tipo de dato  
  
}
```

```
//var myArray = arrayOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)  
var myArray = (0 until 9).toList()  
  
for (i in myArray.indices) {  
    println(myArray[i])  
}  
  
for ((index, value) in myArray.withIndex()) {  
    println("$index.- $value")  
}
```



## 7.- while y do-while

**while:** primero comprueba la condición, es posible que no se ejecute el cuerpo

```
while (quantity < 100) {  
    // instrucciones  
    quantity++  
}
```

**do-while:** la condición se evalúa después de ejecutar el cuerpo, así el cuerpo se ejecuta al menos una vez.

```
do {  
    // instrucciones  
} while (quantity-- > 0)
```

## 8.- repeat

La función **repeat** sirve para repetir el código del cuerpo las veces indicadas, internamente ejecuta un bucle for.

```
repeat( times: 100) { it: Int
|   println("Los 'Billetes Bart' no son de curso legal.")
| }

repeat( times: 100) { index ->
|   println("${index+1}.- Los 'Billetes Bart' no son de curso legal.")
| }
```

## 9.- break, continue y return

Kotlin dispone de las tres maneras típicas de romper el flujo de un bucle.

- **return**: finaliza el bucle y la función donde se encuentre el mismo.
- **break**: finaliza el bucle donde se encuentre.
- **continue**: finaliza la iteración actual del bucle donde se encuentre.

## 9.- break, continue y return

En Kotlin las instrucciones se pueden etiquetar mediante el carácter **@** de la siguiente manera **nombreEtiqueta@** de esta manera se puede romper el flujo de cualquier serie de bucles anidados.

De esta manera poniendo etiquetas, un **break** finalizará el bucle al que esté etiquetando.

```
firstFor@ for (i in 1 ≤ .. ≤ 10) {  
    for (j in 1 ≤ .. ≤ 5) {  
        if (i%2 == 0 && j == 2) break@firstFor  
        println("$i - $j")  
    }  
}
```

# Práctica

## Actividad 3

### Control de flujo en Kotlin

## 10.- Colecciones

Las **colecciones** son un tipo de dato que permiten almacenar un número variable de elementos, cero o más.

Principalmente en una colección se almacenarán elementos del mismo tipo de dato.

Se pueden almacenar datos de diferente tipo pero en este caso se deberá tener especial cuidado cuando se use la colección.

Las colecciones pueden almacenar el valor **null** en alguno de sus elementos.

# 10.- Colecciones

Existen cuatro tipos de colecciones en Kotlin:

## Array:

- Colección de elementos.
- Se accede a los elementos mediante el índice de su posición.
- Se pueden repetir elementos.
- Dos arrays no son iguales aunque contengan los mismos elementos en la misma posición.

## List:

- Colección **ordenada** de elementos.
- Se accede a los elementos mediante el índice de su posición.
- Se pueden repetir elementos.
- Dos List son iguales si contienen los mismos elementos y en la misma posición.

## Set:

- Colección de elementos **sin orden**.
- No se pueden repetir elementos dentro de un Set.
- Dos Set son iguales si contienen los mismos elementos sin importar el orden.

## Map:

- También llamados diccionarios.
- Un Map es un conjunto de pares clave-valor (key-value), no se puede repetir la clave.
- Dos Map son iguales si contienen los mismos pares clave-valor sin importar el orden.

## 10.- Colecciones

Para las List, los Set y los Map, Kotlin dispone de dos versiones.

- **Immutable:**

- No se permite ni añadir ni eliminar elementos a la colección.

- No se permite modificar ningún elemento de la colección.

- **Mutable:**

- Se permite añadir, eliminar y modificar los elementos de la colección.

Sean mutables o no se pueden declarar tanto con **val** como con **var**.

Es su funcionamiento interno el que permite que se puedan modificar o no.



# 11.- List

## Creación de List:

```
// Se indica el tipo de dato de los elementos
val numbers: List<String> = listOf("uno", "dos", "tres", "cuatro", "cinco")
```

```
// Se deduce el tipo de dato de los elementos, todos son iguales → String
val numbers = listOf("uno", "dos", "tres", "cuatro", "cinco")
```

```
//Se indica el tipo de dato
val list: List<Any> = listOf("Rick", 70, "c-137", 'H')
```

```
//Se deduce el tipo de dato. Diferente tipo → Any
val list = listOf("Rick", 70, "c-137", 'H')
```

## Creación de MutableList:

```
// Se indica el tipo de dato pero no se añaden datos
val numbers: MutableList<Int> = mutableListOf()
```

```
// Se indica el tipo de dato de los elementos
val numbers: MutableList<Int> = mutableListOf(1, 2, 3, 4)
```

```
// Se deduce el tipo de dato de los elementos, todos son iguales → Int
val numbers = mutableListOf(1, 2, 3, 4)
```

```
//Se indica el tipo de dato
var list: MutableList<Any> = mutableListOf(1, 'a', "qwerty", true)
```

```
//Se deduce el tipo de dato. Diferente tipo → Any
var list = mutableListOf(1, 'a', "qwerty", true)
```

# 11.- List

## Algunas operaciones sobre List y MutableList

Las operaciones que modifican solo son aplicables a MutableList

```
val list = listOf("uno", "dos", "tres", "cuatro", "cinco")
list.size // Devuelve el valor 5
list[2] // Devuelve la cadena "tres"
list.indexOf("cuatro") // Devuelve el valor 3
list.lastIndexOf(element: "dos") // Devuelve el valor 1
list.subList(1, 3) // Devuelve la lista ["dos", "tres"]

val mutableList = mutableListOf(1, 2, 6, 3, 4, 5)
mutableList.add(6) // añade el elemento 6 al final de la lista
mutableList.add(index: 2, element: 100) // añade el elemento 100 en la posición 2 desplazando el resto a la derecha
mutableList.removeAt(index: 4) // elimina el elemento de la posición 4
mutableList[1] = 99 // cambia el valor de la posición 1 por 99
mutableList.contains(3) // devuelve true/false según se encuentre o no el valor
mutableList.sort() // Ordena ascendentemente los elementos del array
mutableList.sortDescending() // Ordena descendientemente los elementos del array
mutableList.max() // Devuelve el elemento mayor
mutableList.min() // Devuelve el elemento menor
mutableList.isEmpty() // Devuelve true/false según esté vacío o no
mutableList.isNotEmpty() // Devuelve true/false según tenga elementos o no
mutableList.shuffle() // Desordena aleatoriamente los elementos del array
mutableList.shuffled() // Devuelve un array con los elementos del array desordenados
mutableList.first() // Devuelve el primer elemento del array
mutableList.last() // Devuelve el último elemento del array
mutableList.count() // Devuelve la cantidad de elementos del array
mutableList.reverse() // Da la vuelta a los elementos del array
mutableList.reversed() // Devuelve un array con los elementos del array al revés
```

## 12.- Set

### Creación de Set:

```
val names: Set<String> = setOf("Ana", "Jorge", "Lucía")
```

```
val names = setOf("Ana", "Jorge", "Lucía")
```

```
val set: Set<Any> = setOf("Valencia", 2022, "ES", 'V')
```

```
val set = setOf("Valencia", 2022, "ES", 'V')
```

### Creación de MutableSet:

```
val names: MutableSet<String> = mutableSetOf("Ana", "Jorge", "Lucía")
```

```
val names = mutableSetOf("Ana", "Jorge", "Lucía")
```

```
val set: MutableSet<Any> = mutableSetOf("Valencia", 2022, "ES", 'V')
```

```
val set = mutableSetOf("Valencia", 2022, "ES", 'V')
```

# 12.- Set

## Algunas operaciones sobre Set y MutableSet

Las operaciones que modifican solo son aplicables a MutableSet

```
val names = mutableSetOf("Ana", "Jorge", "Lucía")
names.add("Lucas")           // Añade un elemento nuevo
names.add("Ana")             // No añade el elemento al existir ya
names.add("ana")             // Añade el elemento → "ana" != "Ana"
names.remove(element: "Jorge") // Elimina el elemento si se encuentra. Devuelve true/false
names.contains("ana")
names.min()
names.max()
names.isEmpty()
names.isNotEmpty()
names.first()
names.last()
names.reversed()
names.shuffled()
names.isEmpty()
names.isNotEmpty()
names.count()

val set1 = setOf(1, 3, 4, 8, 7, 12, 15)
val set2 = setOf(24, 7, 5, 1, 8, 6)
// Devuelve un Set con los elementos de los dos grupos sin repeticiones
val joinedGroups = set1 union set2
// Devuelve un Set con los elementos comunes a los dos grupos
val commonElements = set1 intersect set2
// Devuelve un Set con los elementos de grupo2 que se encuentren en grupo1
val differentElements = set1 subtract set2
```

# 13.- Map

## Creación de Map:

```
val person: Map<String, String> = mapOf(  
    "name" to "Rick Sanchez",  
    "age" to "70",  
    "estado" to "vivo",  
    "dimension" to "c-137"  
)
```

```
// Se deduce el tipo de dato tanto de la clave como del valor  
val person = mapOf(  
    "name" to "Rick Sanchez",  
    "age" to "70",  
    "estado" to "vivo",  
    "dimension" to "c-137"  
)
```

```
// Map en el que los valores pueden ser de cualquier tipo  
val person: Map<String, Any> = mapOf(  
    "name" to "Rick Sanchez",  
    "age" to "70",  
    "estado" to "vivo",  
    "dimension" to "c-137"  
)
```

## Creación de MutableMap:

```
val person: MutableMap<String, Any> = mutableMapOf(  
    "name" to "Rick Sanchez",  
    "age" to 70,  
    "alive" to true,  
    "dimension" to "c-137"  
)
```

## 13.- Map

### Algunas operaciones sobre Map y MutableMap

Las operaciones que modifican solo son aplicables a MutableMap

```
val person = mutableMapOf(  
    "name" to "Rick Sanchez",  
    "age" to 70,  
    "alive" to true  
)  
person.put("dimension", "c-137") // Añade al Map el par clave-valor indicado  
person.remove(key: "agea") // Elimina y devuelve el elemento que coincide con la clave  
person.keys // Devuelve una lista con las claves  
person.values // Devuelve una lista con los valores  
person.size // Devuelve la cantidad de pares clave-valor  
person.clear() // Vacía el Map  
// Recorrer un Map o MutableMap  
for((key, value) in person) {  
    println("$key -> $value")  
}
```

## 14.- Operaciones sobre colecciones

Algunas operaciones sobre colecciones devuelven un valor booleano que indica si la operación se ha realizado correctamente o no.

Como por ejemplo **add** y **remove**.

Si se quieren conocer todas las operaciones disponibles para **List**, **Set** y **Map** ya sean **mutables** o **solo lectura** se debe consultar la documentación oficial.

## 14.- Operaciones sobre colecciones

Existen operaciones sobre las colecciones que permiten realizar operaciones a todos los elementos de la colección y devuelven una nueva colección con los resultados.

Se podría decir que "recorren" la colección.

Estas operaciones admiten una **función lambda** lo que significa que se deben usar llaves para delimitar las instrucciones. (En la declaración de arrays se vio también el uso de una función lambda, más adelante se explican por completo).

Dentro de la función lambda estará disponible **it** como el elemento actual.



# 14.- Operaciones sobre colecciones

**forEach** recorre todos los elementos de la colección pudiendo realizar acciones sobre ellos.

```
val numbers = listOf(1, 2, 3, 4, 5, 6)
numbers.forEach { it: Int
    | print("${it*-1} ")
    | }
}
```

**filter** devuelve la colección con los elementos que cumplan la condición.

```
// Con Set se realiza de la misma manera
val list = listOf("Uno", "Dos", "Tres", "Cuatro", "Cinco")
val filteredList = list.filter { it: String
    | it.length > 3
    | }
}
```

```
val map = mapOf(1 to "Uno",
                2 to "Dos",
                3 to "Tres",
                4 to "Cuatro",
                5 to "Cinco")
val filteredMap = map.filter { it: Map.Entry<Int, String>
    | it.value.length > 3
    | }
}
```

**map** devuelve la colección aplicando una transformación a cada elemento.

```
// Con Set se realiza de la misma manera
val list = listOf("Uno", "Dos", "Tres", "Cuatro", "Cinco")
val mappedList = list.map { it: String
    | it.uppercase()
    | }
// mappedList contiene la lista con los elementos en mayúsculas
```

```
val map = mapOf(1 to "Uno",
                2 to "Dos",
                3 to "Tres",
                4 to "Cuatro",
                5 to "Cinco")
val mappedMap = map.map { it: Map.Entry<Int, String>
    | it.value.length > 3
    | }
// mappedMap: [false, false, true, true, true]
```

## 15.- Secuencias

Las **secuencias** permiten optimizar las operaciones sobre las colecciones.

```
val list = listOf("Uno", "Dos", "Tres", "Cuatro", "Cinco")  
val secondList = list.filter { it.length > 3 }  
                        .map { it.uppercase() }
```

En el ejemplo anterior las dos operaciones se realizan sobre todos los elementos de la lista generando listas intermedias que no desaparecen hasta acabar todas las operaciones.

Este comportamiento cuando la colección tiene muchos elementos y/o se realizan muchas operaciones sobre la colección va a penalizar en el rendimiento de la aplicación.

## 15.- Secuencias

Para mejorar este comportamiento se puede convertir la colección a una secuencia, realizar todas las operaciones y finalmente volver a generar la colección.

```
val list = listOf("Uno", "Dos", "Tres", "Cuatro", "Cinco")
val secondList = list.asSequence()
                        .filter { it.length > 3 }
                        .map { it.uppercase() }
                        .toList()
```

La secuencia no genera colecciones intermedias.

## 15.- Secuencias

Las secuencias solo generan sus elementos cuando se utilizan por ello permiten generar una cantidad de valores infinitos que posteriormente se pueden utilizar.

Se debe tener cuidado porque se podría bloquear la aplicación esperando a que la secuencia finalice.

```
// Se genera una secuencia infinita de números impares
val oddNumbers = generateSequence( seed: 1) { it + 2 }
// Se pasa por toda la secuencia y se genera una lista con los elementos menores de 200
val oddNumbersTo200 = oddNumbers.takeWhile { it < 200 }.toList()
```

# 16.- Funciones

## Funciones incluidas en el sistema

Como en todo lenguaje de programación, Kotlin incorpora una gran cantidad de funciones ya programadas listas para usar.

Ya se han visto algunas anteriormente: `println`, `toInt`, `toString`...

Por ejemplo, para arrays se tienen disponibles entre otras

<code>min()</code>	<code>max()</code>	<code>isEmpty()</code>	<code>indexOf()</code>
<code>sort()</code>	<code>sorted()</code>	<code>reverse()</code>	<code>reversedArray()</code>

En la documentación se pueden encontrar todas.

## 16.- Funciones

En kotlin las funciones se declaran con la palabra **fun**.

A continuación, se muestra un ejemplo de una función y de su llamada.

```
fun isEven(number: Int): Boolean {  
    return number % 2 == 0  
}
```

```
isEven( number: 16)
```

Las funciones pueden **no recibir parámetros ni devolver valores**.

Si no se indica valor de retorno se devuelve el tipo **Unit** que es equivalente a **void** de otros lenguajes de programación. Si no se devuelve nada se puede omitir **: Unit**.

```
fun greetingPerson(name: String) {  
    println("¡Hola $name!")  
}
```

```
fun greeting() {  
    println("¡Hola!")  
}
```

```
fun greeting(): Unit {  
    println("¡Hola!")  
}
```

## 16.- Funciones

Los parámetros de entrada se separan por comas y se permite el uso de la llamada **trailing comma** (coma final).

```
fun areEqual(  
    num1: Int,  
    num2: Int,  
): Boolean {  
    return if (num1==num2) true else false  
}  
  
areEqual( num1: 26, num2: 123)
```

## 16.- Funciones

**Los parámetros de entrada son inmutables**, no se puede cambiar su valor desde el cuerpo de la función, es como si se declararan con **val**.

Para modificarlos se debe crear una copia.

```
fun addOne(number: Int): Int {  
    number++           // Error: no se puede cambiar el valor  
    return number  
}
```

```
var myNumber = (0 ≤ .. ≤ 100).random()  
myNumber = addOne(myNumber)
```

```
fun addOne(number: Int): Int {  
    val newNumber = number + 1  
    return newNumber  
}
```

```
var myNumber = (0 ≤ .. ≤ 100).random()  
myNumber = addOne(myNumber)
```



## 16.- Funciones

Si la función se puede escribir en una sola línea se **pueden quitar las llaves** y **añadir el símbolo =** antes del cuerpo.

```
fun areEqual(  
    num1: Int,  
    num2: Int,  
): Boolean {  
    return if (num1==num2) true else false  
}  
  
areEqual( num1: 26, num2: 123)
```



```
fun areEqual(  
    num1: Int,  
    num2: Int,  
): Boolean = if (num1==num2) true else false
```

```
fun greeting(nombre: String) = println("¡Hola $nombre!")  
  
fun addition(num1: Int, num2: Int): Int = num1 + num2  
  
fun isEven(num: Int): Boolean = num % 2 == 0  
  
// si no se indica el tipo de dato de retorno Kotlin lo deduce  
fun inRange(num: Int, min: Int, max: Int) = num in min ≤ .. ≤ max  
fun inRange(num: Int, range: IntRange) = num in range
```

## 16.- Funciones

Se puede **usar el nombre de los parámetros en las llamadas a la función** y así cambiar el orden de los parámetros.

Igual que con los intervalos IntelliJ IDEA muestra las pistas en el código (hints) para indicar el nombre del parámetro.

```
fun greeting(  
    name: String,  
    surname: String,  
): String {  
    return "Hola $name $surname"  
}
```

```
greeting("Rick", "Sanchez")  
greeting(name="Rick", "Sanchez")  
greeting("Rick", surname="Sanchez")  
greeting(name="Rick", surname="Sanchez")  
greeting(surname="Sanchez", name="Rick")
```



```
greeting( name: "Rick", surname: "Sanchez")  
greeting(name="Rick", surname: "Sanchez")  
greeting( name: "Rick", surname="Sanchez")  
greeting(name="Rick", surname="Sanchez")  
greeting(surname="Sanchez", name="Rick")
```

## 16.- Funciones

Se permite **definir un valor por defecto** a los parámetros de entrada y de este manera omitir esos parámetros si se quiere.

```
fun finalPrice(  
    price: Double,  
    vat: Int = 21,  
    discount: Int = 0,  
): Double {  
    var finalPrice: Double = price  
    if (discount != 0) finalPrice -= price * discount / 100  
    return finalPrice + finalPrice * vat / 100  
}
```

```
finalPrice( price: 355.0, vat: 10, discount: 5)  
finalPrice( price: 355.0, vat: 10)  
finalPrice( price: 355.0)
```

## 16.- Funciones

Si se usa **valor por defecto** en la definición de la función y **nombres de parámetros en la llamada a la función**, se pueden omitir parámetros intermedios en la llamada.


```
fun finalPrice(  
    price: Double,  
    vat: Int = 21,  
    discount: Int = 0,  
): Double {  
    var finalPrice: Double = price  
    if (discount != 0) finalPrice -= price * discount / 100  
    return finalPrice + finalPrice * vat / 100  
}  
  
finalPrice( price: 355.0, vat: 10, discount: 5)  
finalPrice( price: 355.0, vat: 10)           // descuento → valor por defecto  
finalPrice( price: 355.0, discount = 20)     // iva → valor por defecto
```

## 16.- Funciones

Si se usan parámetros con y sin nombre en la llamada, **los argumentos con nombre deben de ser los últimos** a no ser que ocupen su lugar.

```
greeting( name: "Ana", surname1: "López", surname2: "Sanz")  
greeting(name="Ana", surname1: "López", surname2: "Sanz")  
greeting( name: "Ana", surname2="Sanz", surname1="López")  
greeting(surname2="Sanz", name="Ana", surname1="López")
```

```
greeting( name: "López", surname2="Sanz", name="Ana")
```



# 16.- Funciones

Es posible **declarar un número variable de parámetros**.

Para ello se utiliza un parámetro con la palabra **vararg**.

Habitualmente suele ser el último parámetro. Si no es el último, los siguientes tendrán que pasarse usando su nombre.

Con un **intArray** o una **colección que se pueda convertir a intArray** se puede usar **\*** (operador spread) para pasarlo completamente como parámetro **vararg**.

```
fun max(vararg numbers: Int): Int {  
    var max: Int = numbers[0]  
    for (number in numbers) {  
        if (number > max) max = number  
    }  
    return max  
}
```

```
max( ...numbers: 5, 32, 14, 6, 57, 7, 26, 12)  
val numbers = intArrayOf(5, 32, 14, 6, 57, 7, 26, 12)  
max(*numbers)  
max( ...numbers: 3, 67, 23, *numbers)  
max( ...numbers: 3, 67, 23, *numbers, 42, 17, 92)  
val otherNumbers: Array<Int> = arrayOf(34, 24, 15)  
max( ...numbers: 3, 67, 23, *numbers, 42, 17, 92, *otherNumbers.toIntArray())
```

## 16.- Funciones

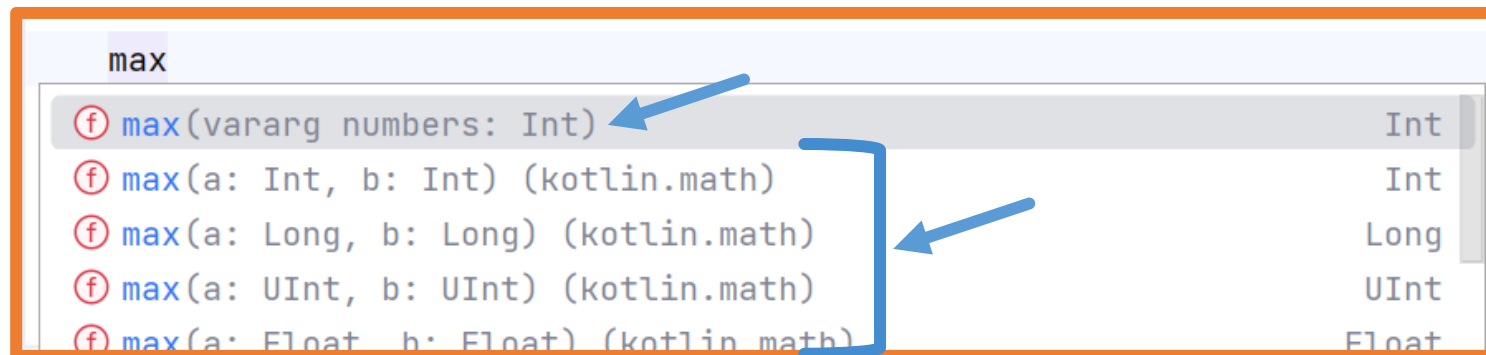
Kotlin permite la **sobrecarga** de funciones aunque estas no estén definidas en una clase.

```
fun addition() = println("No puedo sumar sin valores")

fun addition(num1: Int, num2: Int) = println("$num1 + $num2 = ${num1 + num2}")

fun addition(num1: Double, num2: Double) = println("$num1 + $num2 = ${num1 + num2}")
```

En ejemplos anteriores se ha creado la función **max** que como ya existe se realiza sobrecarga, pudiendo ver la función propia y las del sistema.



## 17.- Ámbito de las funciones

Igual que con las variables el ámbito de una función indica dónde se puede utilizar la función

En Kotlin existen los siguientes tipos de funciones:

- **Nivel superior** (top level)
- **Local**
- **Miembro**
- **De extensión**

Según el tipo se podrá usar en una parte de código u otra.



# 17.- Ámbito de las funciones

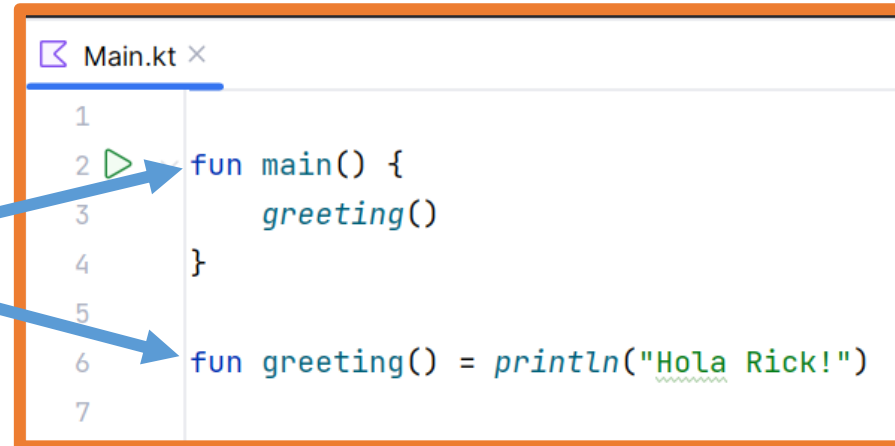
## Nivel superior (top level)

se definen sin estar dentro de ningún bloque { } o clase.

Su uso suele ser como biblioteca de funciones, en este caso en un archivo se incluyen varias funciones de nivel superior que se podrán utilizar en cualquier parte.

Se podrán usar en todo el archivo propio y en el que se importe.

**Top Level**

A screenshot of a code editor window titled 'Main.kt'. The code contains two top-level functions: 'fun main() { greeting() }' and 'fun greeting() = println("Hola Rick!")'. Two blue arrows originate from the 'Top Level' text on the left and point to the 'fun main()' and 'fun greeting()' definitions. The code is as follows:

```
1  
2 fun main() {  
3     greeting()  
4 }  
5  
6 fun greeting() = println("Hola Rick!")  
7
```

# 17.- Ámbito de las funciones

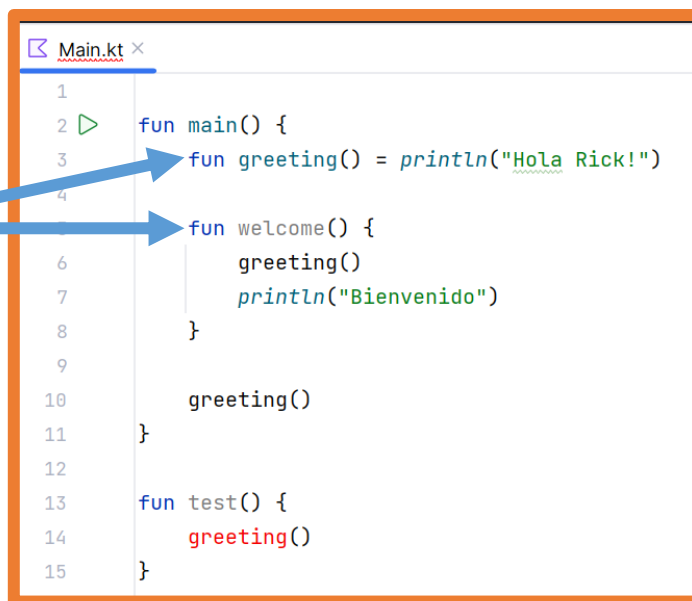
## Local

Se definen dentro de otra función y se podrá usar en todo el bloque { } de esa función, incluso desde otras funciones locales de la misma función padre.

Debe estar definida antes de cualquier llamada a ella.

Pueden usar las variables locales de la función padre.

Local



```
1  
2 fun main() {  
3     fun greeting() = println("Hola Rick!")  
4  
5     fun welcome() {  
6         greeting()  
7         println("Bienvenido")  
8     }  
9  
10    greeting()  
11 }  
12  
13 fun test() {  
14     greeting()  
15 }
```

## 17.- Ámbito de las funciones

### Miembro

Son las funciones que se definen dentro de las clases u objetos.

Son lo que en la programación orientada a objetos se llaman **métodos**.

Se estudian en el punto siguiente **POO** (Programación Orientada a Objetos).

# 17.- Ámbito de las funciones

## De extensión

Kotlin permite **extender la funcionalidad** de las **clases** añadiendo funciones a ellas.

Estas funciones solo existen en el ámbito en el que se definen.

```
fun main() {  
    val number = (0 ≤ .. ≤ 100).random()  
    if(number.isOdd()) println("$number es impar")  
    else println("$number es par")  
}  
  
// Se añade una función a la clase Int  
fun Int.isOdd(): Boolean {  
    return this % 2 != 0  
}
```

# Práctica

## Actividad 4

Técnicas para reutilizar código

## Actividad 5

Arrays en Kotlin