# 77. Lenguaje de programación Kotlin

Kotlin es un lenguaje de programación de tipado estático desarrollado por JetBrains. Fue diseñado para ser completamente interoperable con Java, lo que facilita la transición para los desarrolladores de Java. Kotlin ofrece muchas características modernas como inferencia de tipos, extensiones de función y corutinas, lo que lo hace más conciso y expresivo. Es ampliamente utilizado para el desarrollo de aplicaciones Android, pero también es apto para otros tipos de desarrollo como backend, frontend y aplicaciones nativas. Su seguridad de tipo y diseño sintáctico buscan reducir la cantidad de errores de tiempo de ejecución, mejorando así la calidad del código.

## 77.1 Sitios web para aprender Kotlin

Empezamos por la página principal donde encontramos documentación, referencias, ejercicios y playground para pruebas.

- Documentación que incluye una guía inicial, lo básico y avanzado del lenguaje
- Practicas y ejercicios
- con Kotlin Koans
- Kotlin con ejemplos, ordenados por elementos del lenguaje.
- Curso avanzado de Jetbrain hyperskill (necesita registro. Es gratuito) Se desarrolla por etapas que deben superarse con un test sencillo. Si se desea un certificado del curso, éste es de pago.
- Otros recursos:
- Canal de Slack para Kotlin, por invitación (registro)

## 77.2 Características de Kotlin

- **Interoperabilidad con Java**: Totalmente compatible con Java, lo que permite una fácil transición y uso de bibliotecas existentes.
- **Tipado Estático y Seguro**: Previene errores comunes, como los punteros nulos, mediante un sistema de tipos más seguro.
- **Sintaxis Clara y Concisa**: Reduce la verbosidad del código, facilitando su lectura y mantenimiento.
- **Funciones de Extensión**: Permite añadir nuevas funcionalidades a clases existentes sin tener que modificarlas.
- **Corutinas**: Ofrece soporte nativo para programación asíncrona, lo cual facilita la ejecución de tareas en paralelo.
- **Inferencia de Tipos**: Capaz de deducir automáticamente el tipo de una variable, reduciendo la necesidad de declaraciones explícitas.
- **Inmutabilidad por Defecto**: Fomenta la programación funcional al hacer que las variables sean inmutables por defecto.
- **Smart Casts**: El compilador detecta automáticamente el tipo de objeto, eliminando la necesidad de comprobaciones y conversiones de tipo explícitas.
- **Anotaciones y Delegados de Propiedad**: Ofrecen funcionalidades avanzadas para metaprogramación y gestión eficiente de propiedades.
- **Multiplataforma**: Apto no solo para Android, sino también para el desarrollo de aplicaciones de servidor, web, y más.

## 77.3 Variables y tipos de datos

- `var` : Variable mutable que permite cambios en su valor.
- Ejemplo: `var x = 10`
- `val` : Variable inmutable, equivalente a una constante.
- Ejemplo: `val y = 20`

**Tipos de Datos**

- **Int**: Números enteros.

- Ejemplo: `var a: Int = 5`
- **Double**: Números con decimales.
- Ejemplo: `var b: Double = 5.0`
- **String**: Cadenas de texto.
- Ejemplo: `var c: String = "Hola"`
- **Boolean**: Valores lógicos ( `true` o `false` ).
- Ejemplo: `var d: Boolean = true`

### 77.3.1 Características de las variables en Kotlin

- Kotlin es capaz de inferir el tipo de una variable automáticamente.
- Ejemplo: `val e = "Kotlin"`
- Para hacer que una variable pueda ser `null`, se añade un signo de interrogación al tipo.
- Ejemplo: `var f: String? = null`

### 77.3.2 Tratamiento de null

Kotlin introduce un sistema de tipos nulos para evitar NullPointerException.
Cuando una variable puede ser *null*, se añade "?" al final de la declaración de la variable:
`var a: String? = null`

Disponemos de varios operadores para acceder de forma **segura** a variables y objetos que pueden ser null:

- **Operador de nulabilidad **'?.'** (interrogación punto)
- Ejemplo `val b = a?.length`

- **Operador elvis** '?:' proporciona valor por defecto
- Ejemplo `val c = a?.length ?: 0`
- Operador '!!' fuerza el desenlace de una variable nullable, lanzando NullPointerException si es null.
- Ejemplo : `val c = a !!.length`
- **Operador ?.let{}** Ejecuta una acción solamente si no es nulo.
- Ejemplo

```
    val a: String? = "Hola"
    a?.let {
    println("El valor de a es: $it")
}
```

  La variable **it** dentro del bloque donde se utiliza **?.let** representa el valor no nulo de la variable **a**

## 77.4 Control de flujo de programa

Similar a Java

## 77.5 Funciones kotlin

Las funciones en kotlin se declaran con la palabra **fun**

```
fun doble(x: Int): Int {
    return 2 * x
}
```

Para usar una función.

```
var x = doble(3)
```

### 77.5.1 Parámetros de las funciones

Se declaran con el tipo explícitamente y el tipo devuelto al final:

```
   fun potencia( x: Int, exp: Int, ): Int{}
```

Observar que se puede terminar la lista de parámetros con ","

### 77.5.2 Parámetros por defecto

Se puede indicar un valor por defecto para cualquiera de los parámetros:

```
kotlin
  fun leer(
    b: ByteArray,
    off: Int =0,
    len: Int=b.size
  )
// invocado:
 leer(b)
```

### 77.5.3 Invocar función con parámetros nombrados

Cuando invocamos una función los parámetros se pasan en el mismo orden que se declararon. En kotlin es posible usar el nombre del parámetro al llamar a la función

```
    fun reformat(
     str: String,
     normalizeCase: Boolean = true,
     upperCaseFirstLetter: Boolean = true,
     divideByCamelHumps: Boolean = false,
     wordSeparator: Char = ' ',
) { /*...*/ }
```

Cuando se usa el nombre de los parámetros, estos pueden aparecer en cualquier orden y mezclados con los posicionales:

```
reformat(
    "String!",
    false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
)
```

En este ejemplo también se pueden omitir todos o algunos de los parámetros por defecto

```
    reformat("Esto es un ejemplo ")
```

o bien saltarse alguno de los que tienen valores por defecto. En este caso se debe nombrar todos los parámetros que siguen al que se ha saltado:

```
    reformat("This is a short String!", upperCaseFirstLetter = false, wordSeparator = '_')
```

### 77.5.4 Funciones con retorno **Unit**

Si una función no necesita devolver un valor, siempre devuelve un tipo *Unit*, que tiene un único valor **Unit** de forma implícita o explicita

```
    fun imprimeHola(nombre: String) : Unit {

    }
```

o implícitamente, que es identica a la anterior declaración:

```
fun imprimeHola(nombre: String)  {

    }
```

### 77.5.5 Funciones con una única expresión

En este caso se usa = y se evitan las llaves y **return** :

```
    fun double(x: Int): Int = x * 2
```

También equivalente :

```
    fun double(x: Int) = x * 2
```

Donde se infiere el tipo de dato devuelto.

### 77.5.6 Funciones con un número variable de parámetros.

Se utiliza el modificador **vararg**

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts is an Array
        result.add(t)
    return result
}
```

Dentro de la definición de la función vararg se trata como un *ArrayList*

Se usa con cualquier número de parámetros:

```
   val lista = asList(1,2,3,4)
   val lista2 = asList(19,23)
```

A tener en cuenta: * Sólo se puede marcar un parámetro como *vararg*
* cuando el parámetro *vararg* **NO es** el último, los siguientes deben usarse *nombrados*

Al llamar a la función con vararg se puede pasar un array utilizando el operador "**spread**" (que consiste en usar * delante del nombre del array):

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

### 77.5.7 Notación infix

Con el modificador *infix* las funciones pueden invocarse sin usar el paréntesis de función.

La funciones infix deben cumplir: * Deben ser funciones miembro (métodos) o funciones extensión * Sólo pueden tener un parámetro * El parámetro no puede tener ni valores por defecto ni varargs

```
infix fun Int.shl(x: Int): Int { ... }
```

```
// llamada a una función infix
1 shl 2

// lo mismo que
1.shl(2)
```

### 77.5.8 Funciones operador

Ciertas funciones pueden convertirse en operadores con el modificador "**operator**".

```
operator fun Int.times(str: String) = str.repeat(this)       // 1
println(2 * "Bye ")                                          // 2

operator fun String.get(range: IntRange) = substring(range)  // 3
val str = "Always forgive your enemies; nothing annoys them so much."
println(str[0..14])
```

1. Es una función infix, especial
2. El símbolo para el operador **times** es el asterisco de la multiplicación **"*"** ( ver operator overlodingpara todos los operadores sobrecargables.)

### 77.5.9 Ámbito (Scope) de las funciones

Las funciones en kotlin puede declararse en el nivel más alto sin necesidad de pertenecer a una clase.

Se pueden crear *funciones locales* funciones dentro de otra función

```
fun dfs(graph: Graph) {
    var num_visitados=0

    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)){
```

```
            num_visitados++
            return
        }
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

La función interna puede acceder a las variables de la función contenedora.

### 77.5.10 Funciones genéricas (parametrizadas)

Las funciones genéricas o parametrizadas en Kotlin permiten definir funciones que son independientes respecto al tipo de datos con los que trabajan. En lugar de especificar un tipo de datos concreto, se utiliza un marcador de tipo genérico, normalmente denotado con letras como T, U, V, etc.
Ejemplo:

```
// Definición de una función genérica
fun <T> intercambiar(array: Array<T>, indice1: Int, indice2: Int) {
    val temp: T = array[indice1]
    array[indice1] = array[indice2]
    array[indice2] = temp
}

fun main() {
    val arrayEnteros = arrayOf(1, 2, 3)
    val arrayCadenas = arrayOf("uno", "dos", "tres")

    // Uso de la función genérica con enteros
    intercambiar(arrayEnteros, 0, 2)
    println(arrayEnteros.joinToString(", "))  // Salida: 3, 2, 1

    // Uso de la función genérica con cadenas
    intercambiar(arrayCadenas, 0, 2)
```

```
    println(arrayCadenas.joinToString(", "))  // Salida: tres, dos, uno
}
```

Observar fun delante del nombre de la función.

Ejemplo con varios tipos genéricos:

```
fun <T, U> mezclar(pair: Pair<T, U>): Pair<U, T> {
    return Pair(pair.second, pair.first)
}

fun main() {
    val parIntString = Pair(1, "uno")
    val parStringInt = mezclar(parIntString)

    println("Par original: $parIntString")  // Salida: Par original: (1, uno)
    println("Par mezclado: $parStringInt")  // Salida: Par mezclado: (uno, 1)
}
```

## 77.5.11 Funciones de orden superior y lambdas

### 77.5.11.1 Funciones anónimas y lambda

En muchas ocasiones una función se va a llamar una única vez o en un punto del código concreto.

Casos donde podemos usar funciones anónimas:

Las funciones anónimas y lambdas son **funciones literales** , no se declaran si no que se pasan como expresiones ( no se evaluan para obtener un resultado )

Las funciones anónimas son a menudo útiles cuando trabajas con funciones de orden superior como map, filter, forEach, etc.

```
val numeros = listOf(1, 2, 3, 4)
val cuadrados = numeros.map(fun(x): Int { return x * x })
```

Algunos casos donde usamos funciones lambda/anónimas

- **Manejo de Eventos** Las funciones anónimas son comunes en la programación de interfaces gráficas de usuario para manejar eventos, como clics de botón.

```
boton.setOnClickListener(fun(view: View) {
    // Código para manejar el clic
})
```

- Funciones de libreria, como por ejemplo *sortedBy*

```
val palabras = listOf("manzana", "banana", "cereza")
val ordenadas = palabras.sortedWith(fun(a, b): Int { return a.length - b.length })
```

- Hilos y concurrencias (mas adelante)

Las funciones anónimas y las funciones lambdas tienen semejanzas y diferencias:

#### 77.5.11.2 Funciones lambda

- Sintaxis más concisa: Las funciones lambda tienen una sintaxis más corta y son generalmente más legibles para operaciones simples.

```
val suma = { a: Int, b: Int -> a + b }
```

- Por lo general el compilador kotlin puede inferir el tipo de los parámetros y valor de retorno
- Las funciones lambda pueden acceder a variables externas en el ambito cercano pero **NO pueden modificarlas**.
- Las funciones lambda deben ser cortas y no admiten sentencias if, while, etc

La sintaxis de una función lambda es la siguiente:

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

* Las funciones lambda siempre están rodeadas por llaves {} * La declaración de parámetros se hace dentro de las llaves y pueden tener anotaciones. * el curpo de la función van despues de **->** * Si el retorno de la función no es *Unit* el resultado de la última expresión es el retorno de la función

#### 77.5.11.3 Funciones anónimas

- Sintaxis parecida a las funciones normales
- Tipos explícitos en los parámetros que facilitan la lectura.
- Pueden acceder a variables externas del ámbito cercano
- Pueden tener multiples sentencias en un bloque {}

Ejemplo comparativo:

```
// Función lambda para sumar
val sumaLambda = { a: Int, b: Int -> a + b }

// Función anónima para sumar
val sumaAnonima = fun(a: Int, b: Int): Int { return a + b }
```

#### 77.5.11.4 Convención sintáctica cuando las funciones lambda es el último parámetro en una función de orden superior

En estos casos, por convención, la función lambda puede escribirse fuera del paréntesis:

```
val product = items.fold(1) { acc, e -> acc * e }
```

Es exactamente lo mismo que:

```
val product = items.fold(1, { acc, e -> acc * e } )
```

y corresponde a la declaración de una función de la clase

```
 fun <T, R> Iterable<T>.fold(
    initial: R,
    operation: (acc: R, T) -> R
): R
```

Y en el caso de que la función lambda sea el único parámetro, se puede omitir totalmente:

```
run { println("...") }
// o lo mismo.
run( {println("...")})
```

Que se declara como:

```
inline fun <R> run(block: () -> R): R {
    return block()
}
```

Esto se conoce como "**trailing lambda**"

#### 77.5.11.5 it: nombre implicito cuando hay un único parámetro

Si el compilador puede tratar la signature sin declarar ningún parámetro, entonces de puede omitir este.

```
ints.filter { it > 0 } // this literal is of type '(it: Int) -> Boolean'
```

#### 77.5.11.6 Funciones de orden superior

Una función de orden superior admite como parámetro otra función ( o devuelve una función).

Ejemplo, la siguiente función recibe dos enteror y la operación sobre esos enteror

```
fun calculate(x: Int, y: Int, miOperacion: (Int, Int) -> Int): Int {  // 1
    return miOperacion(x, y)                                          // 2
}

fun sum(x: Int, y: Int) = x + y                                       // 3

fun main() {
    val sumResult = calculate(4, 5, ::sum)                            // 4
    val mulResult = calculate(4, 5) { a, b -> a * b }                 // 5
    println("sumResult $sumResult, mulResult $mulResult")
```

- La función que se pasa en el parámetro se declara indicando su firma (signature) sin nombre de función `(Int, Int) -> Int`
- Como todos los parámetros, tiene un nombre *miOperacion*
- La función que se puede usar como parámetro debe cumplir la signature declarada en la función superior
- Y cuando se llama se usa la notación "**::sum**"

La signatura de la función también se puede hacer como función anónima:

```
// 1. Declaramos 'calculate' como una función de orden superior que toma otra función 'miOperacion' como parámetro
fun calculate(x: Int, y: Int, miOperacion: fun(Int, Int): Int): Int {
    // 2. Invocamos 'miOperacion' con los argumentos 'x' y 'y' y retornamos el resultado
    return miOperacion(x, y)
}

// Uso de la función 'calculate'
fun main() {
    // Suma
    val resultadoSuma = calculate(5, 3, fun(a: Int, b: Int): Int { return a + b })
    println("Resultado de la suma: $resultadoSuma")  // Salida: Resultado de la suma: 8

    // Multiplicación
```

```
    val resultadoMultiplicacion = calculate(5, 3, fun(a: Int, b: Int): Int { return a * b })
    println("Resultado de la multiplicación: $resultadoMultiplicacion")  // Salida: Resultado de la multiplicación: 15
}
```

Ejemplo donde se devuelve una función:

```
fun operation(): (Int) -> Int {
    return ::square
}

fun square(x: Int) = x * x

fun main() {
    val func = operation()
    println(func(2))
}
```

## 77.6 Otros enlaces

- Descarga de JetBrains Intellidea education edition

**¿Fue útil esta página?**