

Funciones y lambdas en Kotlin: Crea código reutilizable y elegante



- [1. Introducción](#)
- [2. ¿Qué son las funciones y cómo se declaran?](#)
- [3. Parámetros nombrados](#)
- [4. Valores por defecto](#)
- [5. Funciones que no devuelven nada](#)
- [6. Funciones de una sola expresión](#)
- [7. Retornos tempranos en funciones](#)
- [8. Expresiones lambda](#)
 - [8.1. Pasando lambdas a otras funciones](#)
 - [8.2. Tipos de funciones](#)
- [9. Ejercicios](#)
 - [9.1. El área del círculo](#)
 - [9.2. El área del círculo en una sola línea](#)
 - [9.3. Pasando el tiempo a segundos](#)
 - [9.4. Generador de URLs](#)
 - [9.5. Repetidor de acciones](#)
 - [9.6. El saludo personalizado](#)
 - [9.7. Filtrando palabras largas](#)
- [10. Soluciones a los ejercicios](#)
 - [10.1. El área del círculo](#)
 - [10.2. El área del círculo en una sola línea](#)
 - [10.3. Pasando el tiempo a segundos](#)
 - [10.4. Generador de URLs](#)
 - [10.5. Repetidor de acciones](#)
 - [10.6. El saludo personalizado](#)
 - [10.7. Filtrando palabras largas](#)

1. Introducción

Si has llegado hasta aquí, ya sabes cómo crear variables, manejar distintos tipos de datos, organizar información en colecciones y controlar el flujo de tu programa.

con condicionales y bucles. ¡Enhorabuena! Tienes las bases. Pero ahora toca dar el siguiente paso lógico en la programación: **las funciones**.

Imagina que en tu código tienes que hacer un cálculo complejo o mostrar un mensaje específico docenas de veces a lo largo del programa. ¿Vas a copiar y pegar el mismo bloque de código una y otra vez? Para eso están las funciones. En esta unidad, vamos a ver cómo Kotlin hace que declarar y usar funciones sea un proceso muy limpio, adentrándonos por último en el fascinante mundo de las **expresiones lambda**, una de las características más apreciadas por los desarrolladores modernos.

2. ¿Qué son las funciones y cómo se declaran?

En Kotlin, puedes declarar tus propias funciones utilizando la palabra reservada `fun` (muy apropiado, ¿verdad? ¡Programar en Kotlin es divertido!). Una función no es más que un bloque de código al que le damos un nombre y que realiza una tarea concreta.

```
1. fun sumar(x: Int, y: Int): Int {  
2.     return x + y  
3. }  
4.  
5. fun main() {  
6.     println(sumar(1, 2))  
7.     // Salida: 3  
8. }
```

Analizando el código paso a paso:

- **fun**: La palabra clave mágica para decirle a Kotlin que vamos a crear una función.
- **Los parámetros**: Se escriben siempre entre paréntesis `()`. Cada parámetro debe tener un tipo de dato explícito (en este caso `x` e `y` son de tipo `Int`), y si hay varios, se separan por comas. Son los datos que la función necesita para poder trabajar.
- **El tipo de retorno**: Despues de los paréntesis de los parámetros y separado por dos puntos `:`, indicamos qué tipo de dato va a devolver la función como resultado final. En este caso, devolverá un número entero (`Int`).

- **El cuerpo de la función:** Es todo lo que va dentro de las llaves {} .
- **return :** Usamos esta palabra para escupir o devolver el resultado de nuestro cálculo y dar por terminada la ejecución de la función.

El dato: Las convenciones de código oficiales de Kotlin recomiendan nombrar las funciones empezando con minúscula y usando camelCase (por ejemplo, calcularPrecioTotal, imprimirUsuario), sin usar guiones bajos.

3. Parámetros nombrados

A veces, cuando llamas a una función que tiene muchísimos parámetros, el código puede volverse un poco lioso a simple vista. ¿Qué era ese true o ese 5 que le pasábamos a la función en la tercera posición? Kotlin nos permite usar **parámetros nombrados**.

Si incluyes el nombre del parámetro explícitamente al llamar a la función, no solo haces que tu código sea muchísimo más fácil de leer, sino que además **puedes poner los parámetros en el orden que quieras**.

```
1. fun imprimirMensaje(mensaje: String, prefijo: String) {  
2.     println("[\$prefijo] \$mensaje")  
3. }  
4.  
5. fun main() {  
6.     // Usamos argumentos nombrados y además invertimos el orden  
7.     imprimirMensaje(prefijo = "LOG", mensaje = "Sistema iniciado  
correctamente")  
8.     // Resultado: [LOG] Sistema iniciado correctamente  
9. }
```

4. Valores por defecto

En lenguajes más antiguos, si a veces querías llamar a una función pasándole tres parámetros y otras veces solo dos, tenías que crear varias versiones de la misma función (esto se conoce como sobrecarga de métodos).

En Kotlin, te ahorras todo ese trabajo. Puedes definir **valores por defecto** para los parámetros utilizando el operador de asignación `=`. Si al llamar a la función omites ese parámetro, Kotlin utilizará automáticamente el valor por defecto que configuraste.

```
1. fun imprimirMensaje(mensaje: String, prefijo: String = "INFO") {  
2.     println("[\$prefijo] \$mensaje")  
3. }  
4.  
5. fun main() {  
6.     // Llamamos a la función pasándole ambos parámetros  
7.     imprimirMensaje("Usuario conectado", "LOG")  
8.     // Resultado: [LOG] Usuario conectado  
9.  
10.    // Llamamos a la función pasándole SOLO el mensaje. ¡Usa el  
11.    // prefijo por defecto!  
12.    imprimirMensaje("El proceso ha finalizado")  
13. }
```

La regla de oro: Puedes saltarte parámetros concretos que tengan valores por defecto, pero si omites uno y quieres pasar el siguiente, tendrás que usar obligatoriamente parámetros nombrados para los que pongas a continuación.

5. Funciones que no devuelven nada

Si tu función simplemente realiza una acción (como imprimir un texto por la consola, reproducir un sonido o guardar un dato en la base de datos) pero no necesita devolverte ningún resultado matemático o lógico útil, su tipo de retorno en Kotlin es `Unit` (el equivalente al `void` en Java o C).

La maravilla de Kotlin es que **no hace falta que escribas `Unit` ni que pongas un `return`**. El compilador ya lo sabe.

```
1. fun saludar(nombre: String) {  
2.     println("¡Hola, $nombre!")  
3.     // Escribir `: Unit` arriba junto a los paréntesis o `return  
4.     // Unit` aquí abajo es totalmente opcional (y casi nadie lo hace).  
5. }
```

6. Funciones de una sola expresión

Para hacer tu código aún más conciso y espectacular, si tu función consta de una única instrucción que devuelve un valor, puedes ahorrarte de un plumazo las llaves {} y la palabra `return`. Solo tienes que usar el signo igual = .

Tomemos la función `sumar` del principio:

```
1. fun sumar(x: Int, y: Int): Int {  
2.     return x + y  
3. }
```

Se puede transformar en esta maravilla de una sola línea:

```
1. fun sumar(x: Int, y: Int) = x + y
```

Fíjate bien, ¡ni siquiera hemos puesto : Int ! Como Kotlin cuenta con una característica llamada **inferencia de tipos**, deduce inmediatamente que si sumas dos `Int`, el resultado será forzosamente un `Int`.

(No obstante, si estás trabajando en equipo y quieres que otros programadores lean tu código rápidamente, nunca está de más indicar explícitamente el tipo de retorno: `fun sumar(x: Int, y: Int): Int = x + y`).

7. Retornos tempranos en funciones

En ocasiones, quieres salir de una función inmediatamente si se cumple (o no) una condición, sin necesidad de seguir ejecutando y procesando el resto del código. Para eso usamos la palabra `return` dentro de un condicional `if` .

```
1. val usuariosRegistrados = mutableListOf("juan_perez", "maria_lopez")  
2.  
3. fun registrarUsuario(usuario: String): String {  
4.     // Retorno temprano si el usuario ya existe en nuestra base de  
    datos (lista)  
5.     if (usuario in usuariosRegistrados) {  
6.         return "Error: El nombre de usuario ya está pillado. Elige  
    otro."  
7.     }  
8. }
```

```

9.      // Si no hemos entrado en el if anterior, la función continúa de
10.     forma normal
11.     usuariosRegistrados.add(usuario)
12.     return "¡Usuario $usuario registrado con éxito!"
13.
14. fun main() {
15.     println(registrarUsuario("juan_perez"))
16.     // Salida: Error: El nombre de usuario ya está pillado. Elige
17.     otro.
18.     println(registrarUsuario("nuevo_user"))
19.     // Salida: ¡Usuario nuevo_user registrado con éxito!
20. }

```

8. Expresiones lambda

Kotlin te permite escribir funciones de forma todavía más compacta usando lo que conocemos como **expresiones lambda**. Básicamente, una lambda es una función anónima (sin nombre) que puedes guardar directamente en una variable o pasarlal como si fuera un parámetro más a otras funciones.

Mira esta función normal:

```

1. fun aMayusculas(texto: String): String {
2.     return texto.uppercase()
3. }

```

Podemos escribir exactamente lo mismo como una expresión lambda y guardarla en una variable:

```

1. val aMayusculasLambda = { texto: String -> texto.uppercase() }
2.
3. fun main() {
4.     println(aMayusculasLambda("hola"))
5.     // Resultado: HOLA
6. }

```

Estructura de una lambda:

- Se encapsula siempre entre llaves { } .
- Primero van los parámetros (texto: String).

- Luego una flecha `->`.
- Y finalmente el cuerpo de la función (lo que hace y lo que se devuelve de forma automática: `texto.uppercase()`).

8.1. Pasando lambdas a otras funciones

Esto resulta muy útil cuando trabajamos con colecciones (listas, sets, mapas...). Multitud de funciones nativas de colecciones reciben una lambda para saber exactamente qué deben hacer con cada uno de los elementos.

El mejor ejemplo es usar `.filter()` (para filtrar elementos en base a una condición) o `.map()` (para transformar los elementos de una lista en otros):

```

1. fun main() {
2.     val numeros = listOf(1, -2, 3, -4, 5, -6)
3.
4.     // Quedarnos solo con los positivos. La lambda comprueba si el
5.     // número es mayor que 0.
6.     val positivos = numeros.filter({ x -> x > 0 })
7.     println(positivos) // [1, 3, 5]
8.
9.     // Multiplicar todos los elementos de la lista por 2 usando map
10.    val dobles = numeros.map({ x -> x * 2 })
11.    println(dobles) // [2, -4, 6, -8, 10, -12]
12. }
```

Si la lambda es el último o el único parámetro que le pasas a una función (como pasa arriba con filter y map), Kotlin te permite sacarla fuera de los paréntesis `()`. E incluso puedes omitir los paréntesis por completo si es el único argumento.

Reescribiendo lo anterior de forma idiomática:

```
1. val positivos = numeros.filter { x -> x > 0 }
```

8.2. Tipos de funciones

Al igual que una variable puede ser de tipo `String` o `Int`, ¡las funciones también tienen su propio tipo! La sintaxis para definir el tipo de una función consiste en poner los parámetros de entrada entre paréntesis y, tras una flecha `->`, el tipo que va a devolver.

- `(String) -> String` (recibe un texto, devuelve un texto)
- `(Int, Int) -> Int` (recibe dos enteros, devuelve un entero)
- `() -> Unit` (no recibe absolutamente nada, no devuelve nada útil)

Ejemplo declarando el tipo de forma explícita en una variable:

```
1. val sumarLambda: (Int, Int) -> Int = { a, b -> a + b }
```

9. Ejercicios

Aquí tienes varios ejercicios de menos a más dificultad (algunos adaptados de la documentación oficial y otros extra para asentar todo lo aprendido). Abre tu [Kotlin Playground](#) y vamos a practicar lo aprendido.

9.1. El área del círculo

Escribe una función normal llamada `areaCirculo` que reciba el radio de un círculo en formato entero (`Int`) y devuelva el área de ese círculo (con decimales).

Pista: Vas a necesitar importar la constante Pi desde `kotlin.math.PI`. La fórmula es `PI * radio * radio`.

```
1. import kotlin.math.PI
2. // Escribe tu función aquí
3.
4. fun main() {
5.     println(areaCirculo(2)) // Debería dar aprox 12.566...
6. }
```

9.2. El área del círculo en una sola línea

Reescribe la función `areaCirculo` del ejercicio anterior pero simplifícalo para convertirla en una **función de una sola expresión** (Single-expression function), eliminando las llaves y el `return`.

9.3. Pasando el tiempo a segundos

Tienes una función que traduce un intervalo de tiempo (horas, minutos y segundos) a segundos totales. La inmensa mayoría de las veces, solo vas a querer pasarle un parámetro y que el resto sean cero por defecto.

Mejora la función y las llamadas en el `main` utilizando **valores por defecto** y **argumentos nombrados** para que el código quede impecable y súper legible.

```
1. // Mejora esta función
2. fun intervaloEnSegundos(horas: Int, minutos: Int, segundos: Int) =
3.     ((horas * 60) + minutos) * 60 + segundos
4.
5. fun main() {
6.     // Mejora estas llamadas
7.     println(intervaloEnSegundos(1, 20, 15))
8.     println(intervaloEnSegundos(0, 1, 25))
9.     println(intervaloEnSegundos(2, 0, 0))
10.    println(intervaloEnSegundos(0, 10, 0))
11.    println(intervaloEnSegundos(1, 0, 1))
12. }
```

9.4. Generador de URLs

Tienes una lista de acciones de una API, un prefijo de una web y el ID de un recurso. Utilizando la función de colección `.map()` y una **expresión lambda**, crea una nueva lista de URLs que combinen los tres elementos para formar rutas completas (ejemplo esperado: `https://ejemplo.com/libro/5/titulo`).

```
1. fun main() {
2.     val acciones = listOf("titulo", "anyo", "autor")
3.     val prefijo = "https://ejemplo.com/libro"
4.     val id = 5
5. }
```

```
6.     val urls = // ¡Escribe aquí tu map con lambda!
7.
8.     println(urls)
9. }
```

9.5. Repetidor de acciones

Escribe una función llamada `repetirN` que reciba un número entero `n` y una acción (una función de tipo `() -> Unit`). La función debe repetir esa acción el número de veces indicado utilizando un bucle clásico. Luego, en el `main`, úsala (usando una *trailing lambda*) para imprimir "¡Me encanta Kotlin!" 5 veces.

9.6. El saludo personalizado

Crea una función llamada `saludoPersonalizado` que reciba un `nombre` (`String`), una `edad` (`Int`) y una `ciudad` (`String`). El parámetro `ciudad` debe tener el valor por defecto "Madrid".

La función debe usar un `println()` para mostrar: "Hola, me llamo [nombre], tengo [edad] años y vivo en [ciudad].".

Luego, en el `main`, llama a la función de dos formas distintas:

1. Pasando solo nombre y edad.
2. Pasando nombre, edad y ciudad, pero utilizando **argumentos nombrados** en un orden distinto al original.

9.7. Filtrando palabras largas

Dada la siguiente lista de palabras: `val palabras = listOf("sol", "murciélagos", "pan", "ornitorrinco", "luz", "desarrollador")`

Utiliza la función `.filter()` y una **trailing lambda** para crear una nueva lista que solo contenga las palabras que tengan estrictamente más de 5 letras. (Pista: Los

textos o `Strings` en Kotlin tienen una propiedad llamada `.length` que te dice lo largos que son). Imprime el resultado final.

10. Soluciones a los ejercicios

No hagas trampas. Échale un vistazo a las soluciones únicamente cuando te hayas peleado un buen rato con el código.

10.1. El área del círculo

```
1. import kotlin.math.PI
2.
3. fun areaCirculo(radio: Int): Double {
4.     return PI * radio * radio
5. }
6.
7. fun main() {
8.     println(areaCirculo(2))
9.     // 12.566370614359172
10. }
```

10.2. El área del círculo en una sola línea

```
1. import kotlin.math.PI
2.
3. // Quitamos llaves y return, y usamos el signo '='. Mantener el ': Double' es buena práctica.
4. fun areaCirculo(radio: Int): Double = PI * radio * radio
5.
6. fun main() {
7.     println(areaCirculo(2))
8. }
```

10.3. Pasando el tiempo a segundos

```
1. // Asignamos '= 0' a todos los parámetros para darles un valor por defecto
2. fun intervaloEnSegundos(horas: Int = 0, minutos: Int = 0, segundos: Int = 0) =
```

```

3.     ((horas * 60) + minutos) * 60 + segundos
4.
5.     fun main() {
6.         println(intervaloEnSegundos(1, 20, 15)) // Este lo pasamos normal
7.         println(intervaloEnSegundos(minutos = 1, segundos = 25)) //
Omitimos las horas
8.         println(intervaloEnSegundos(horas = 2)) // Omitimos minutos y
segundos
9.         println(intervaloEnSegundos(minutos = 10))
10.        println(intervaloEnSegundos(horas = 1, segundos = 1))
11.    }

```

10.4. Generador de URLs

```

1.     fun main() {
2.         val acciones = listOf("titulo", "anyo", "autor")
3.         val prefijo = "https://ejemplo.com/libro"
4.         val id = 5
5.
6.         // Usamos map para transformar cada elemento. Recuerda usar
plantillas de texto ($) para fusionarlo todo.
7.         val urls = acciones.map { accion -> "$prefijo/$id/$accion" }
8.
9.         println(urls)
10.        // [https://ejemplo.com/libro/5/titulo,
https://ejemplo.com/libro/5/anyo, https://ejemplo.com/libro/5/autor]
11.    }

```

10.5. Repetidor de acciones

```

1.     // El tipo de la acción es () -> Unit porque es un bloque de código
que no recibe nada y no devuelve nada útil.
2.     fun repetirN(n: Int, accion: () -> Unit) {
3.         for (i in 1..n) {
4.             accion()
5.         }
6.     }
7.
8.     fun main() {
9.         // Como la acción es el último parámetro, podemos sacar las
llaves (trailing lambda)
10.        repetirN(5) {
11.            println("¡Me encanta Kotlin!")
12.        }
13.    }

```

10.6. El saludo personalizado

```
1. fun saludoPersonalizado(nombre: String, edad: Int, ciudad: String =  
    "Madrid") {  
2.     println("Hola, me llamo $nombre, tengo $edad años y vivo en  
    $ciudad.")  
3. }  
4.  
5. fun main() {  
6.     // Llamada 1: usamos el valor por defecto para ciudad  
7.     saludoPersonalizado("Carlos", 28)  
8.  
9.     // Llamada 2: usamos argumentos nombrados alterando el orden  
    natural  
10.    saludoPersonalizado(ciudad = "Valencia", edad = 32, nombre =  
        "Laura")  
11. }
```

10.7. Filtrando palabras largas

```
1. fun main() {  
2.     val palabras = listOf("sol", "murciélagos", "pan", "ornitorrinco",  
    "luz", "desarrollador")  
3.  
4.     // filter con trailing lambda. Solo conservamos la palabra si su  
    longitud es mayor a 5.  
5.     val palabrasLargas = palabras.filter { palabra -> palabra.length  
    > 5 }  
6.  
7.     println(palabrasLargas)  
8.     // [murciélagos, ornitorrinco, desarrollador]  
9. }
```