



UD2.4 – Fundamentos de Kotlin

2º CFGS
Desarrollo de Aplicaciones Multiplataforma
2023-24

1.- Funciones de alcance

Kotlin ofrece las llamadas **Scope functions** que permiten ejecutar un bloque de código **en el contexto del objeto que las llama**.

Al ejecutarse en el contexto del objeto que las llama, dentro del cuerpo de la función está disponible dicho objeto.

Existen cinco funciones de alcance:

- let
- run
- with
- apply
- also

1.- Funciones de alcance

Función	Uso	Contexto	Devuelve
let	Ejecutar bloque de código asegurándose de que el objeto no es null.	Propio objeto accesible con it	Resultado de la última instrucción
apply	Configuración de un objeto.	Propio objeto accesible con this	Propio objeto
run	Configuración de un objeto y ejecución de instrucciones sobre él.	Propio objeto accesible con this	Resultado de la última instrucción
run	Ejecutar instrucciones cuando se requiere una expresión, se usa sin que sea llamada desde un objeto.	-	Resultado de la última instrucción
also	Seguir realizando instrucciones sobre el objeto.	Propio objeto accesible con it	Propio objeto
with	Agrupar llamadas a funciones de un objeto.	Propio objeto accesible con this	Resultado de la última instrucción

Estas funciones devuelven un valor pero no es necesario capturarlo (guardarlo) en ninguna variable.

1.- Funciones de alcance

Todas las funciones de alcance se utilizan con un bloque de llaves { } (función lambda).

IntelliJ IDEA muestra las pistas en el código (hints) para ayudar a entender cómo funciona cada una de ellas.

```
class Product(var name: String, var price: Double, var year: Int)

val product = Product( name: "PS5", price: 495.50, year: 2021)
var result = product.let { it: Product
    println("Se va a aplicar un descuento")

    it.price * 0.85 ^let
}
```

Contexto

^let: la flecha indica que se devuelve el resultado

```
class Product(var name: String, var price: Double, var year: Int)

val product = Product( name: "PS5", price: 495.50, year: 2021)
var result = product.let { it: Product
    println("Si el producto es antiguo se aplicará un descuento")
    if (it.year < 2023) it.price * 0.85 ^let
    else it.price ^let
}
```

Contexto

^let: como hay 2 caminos posibles aparece 2 veces

2.- let

let

Contexto: el objeto desde el cual se llama, accesible con **it**.

Devuelve: el resultado de la última instrucción.

```
class Product(var name: String, var price: Double, var year: Int)

val product: Product = Product(name: "PS5", price: 495.50, year: 2021)
var result = product.let { it: Product
    println("Si el producto es antiguo se aplica un descuento")
    if (it.year < 2023) it.price * 0.85 ^let
    else it.price ^let
}
```

2.- let

let

El uso de let asegura que el objeto no será null antes de ejecutar las instrucciones.

```
class Product(var name: String, var price: Double, var year: Int)

var product: Product? = Product(name: "PS5", price: 495.50, year: 2021)
val number = (1..100).random()
if (number % 2 == 0) product = null
var result = product?.let { it: Product
    println("Si el producto es antiguo se aplica un descuento")
    if (it.year < 2023) it.price * 0.85 ^let
    else it.price ^let
}
```

En el caso de que **el objeto producto** sea null el bloque **let** no se ejecutará.

3.- run

run

Tiene dos usos:

- 1.- Llamada desde un objeto como let, pero no controla si la variable es null.

El contexto: el objeto desde el cual se llama, accesible con **this**. No es necesario poner **this**. para acceder a las propiedades.

Devuelve: el resultado de la última instrucción.

```
class Product(var name: String, var price: Double, var year: Int) {  
    fun addPercentageToPrice(percentage: Double) {  
        price += price * percentage / 100  
    }  
}  
  
val product: Product = Product( name: "PS5", price: 495.50, year: 2021)  
val result = product.run { this: Product  
    // this.price /=2  
    name += " [OFERTA]"  
    if (price < 1000.0) addPercentageToPrice( percentage: 12.5)  
    price ^run  
}  
println("Producto: ${product.name} ${product.price}")  
println(result)
```

Salida:

```
Producto: PS5 [OFERTA] 557.4375  
557.4375
```

3.- run

run

Tiene dos usos:

2.- Llamada sin usar objeto.

El contexto: no hay contexto.

Devuelve: el resultado de la última instrucción.

```
val name = "Rick Sanchez"
val containsR = run {
    var letterR = false
    for (letter in name) {
        if (letter == 'R') {
            letterR = true
            break
        }
    }
    letterR ^run
}
```


4.- also

also

Permite realizar acciones extra (also = además) sobre el objeto que la llama.

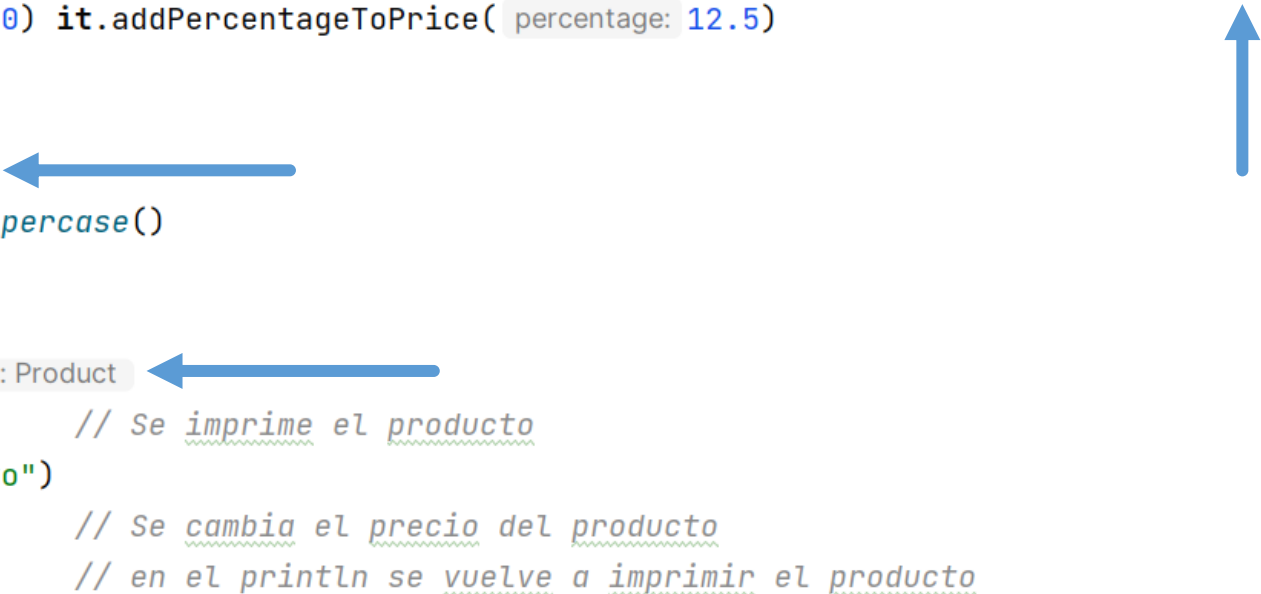
El contexto: el objeto desde el cual se llama, accesible con **it**.

Devuelve: el propio objeto (se hace automáticamente).

```
val product: Product = Product( name: "Nintendo Switch", price: 350.0, year: 2021).also { it: Product
    if (it.price < 1000.0) it.addPercentageToPrice( percentage: 12.5)
}

product.also { it: Product
    it.name = it.name.uppercase()
}

println(product.also { it: Product
    println("---> $it")    // Se imprime el producto
    println("Precio nuevo")
    it.price *= 0.85      // Se cambia el precio del producto
                          // en el println se vuelve a imprimir el producto
})
```



5.- apply

apply

Se utiliza para configurar (asignar valores) a un objeto.

El contexto: el objeto desde el cual se llama, accesible con **this**.

Devuelve: el propio objeto (se hace automáticamente).

```
val product: Product = Product( name: "Nintendo Switch", price: 300.0, year: 2021)
product.apply { this: Product ←
    // this.name = "Switch OLED"
    name = "Switch OLED"
    price = 349.90
}
```

6.- with

with

Permite agrupar acciones sobre un objeto.

El contexto: el objeto desde el cual se llama, accesible con **this**.

Devuelve: el resultado de la última instrucción.

```
val product: Product = Product( name: "Nintendo Switch", price: 300.0, year: 2021)
with(product) { this: Product
    println(
        """$name
        | Precio anterior: $price €
        """.trimMargin()
    )
    if (price < 1000.0) addPercentageToPrice( percentage: 12.5)
    println(" Precio nuevo: $price €")

    println(this)
}
```

6.- with

with

Permite agrupar acciones sobre un objeto.

El contexto: el objeto desde el cual se llama, accesible con **this**.

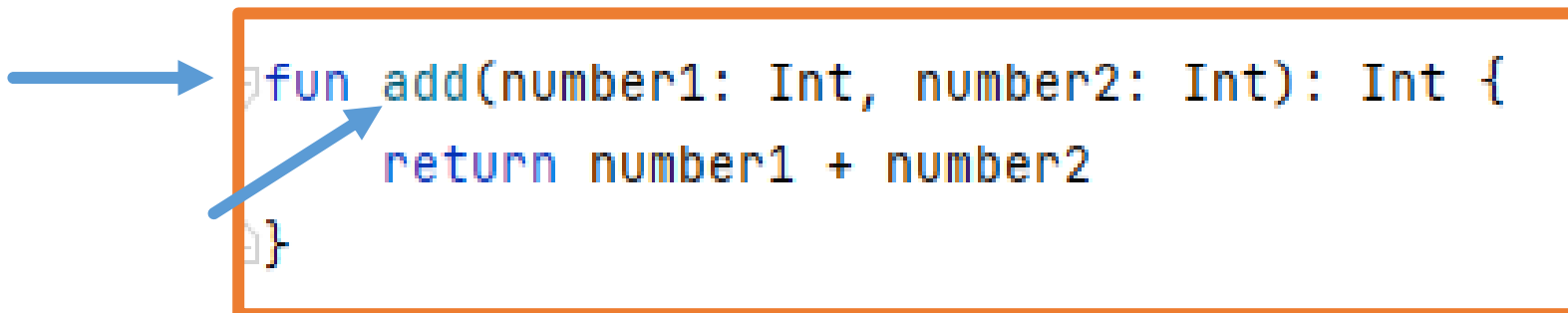
Devuelve: el resultado de la última instrucción.

```
val product: Product = Product( name: "Nintendo Switch", price: 300.0, year: 2021)
val newProduct = with(product) { this: Product ←
    println(
        """$name
        | Precio anterior: $price €
        """.trimMargin()
    )
    if (price < 1000.0) addPercentageToPrice( percentage: 12.5)
    println(" Precio nuevo: $price €")

    println(this)
    this ^with ←
}
```

7.- Funciones lambda

Como norma general una función se debe declarar y se le debe asignar un identificador para poder usarse:



```
fun add(number1: Int, number2: Int): Int {  
    return number1 + number2  
}
```

The diagram shows a code block with an orange border. A blue arrow points from the left to the word 'fun'. Another blue arrow points from the left to the parameter 'number1'.

Las funciones **lambda**, también llamadas funciones anónimas o funciones flecha son funciones que **no están declaradas (no tienen identificador) y se utilizan como una expresión.**

7.- Funciones lambda


Las funciones lambda **se escriben entre llaves { }**, pueden tener parámetros o no y deben tener un cuerpo.

Si no tiene parámetros se deben poner directamente las instrucciones.

El resultado de la última instrucción de una lambda se devolverá, si la función tiene varias instrucciones IntelliJ IDEA lo indica con las pistas en el código (hints).

```
{ println("Desde una lambda") }
```

```
{  
    var text = "Desde una lambda"  
    text += "!"  
    text ^lambda  
}
```

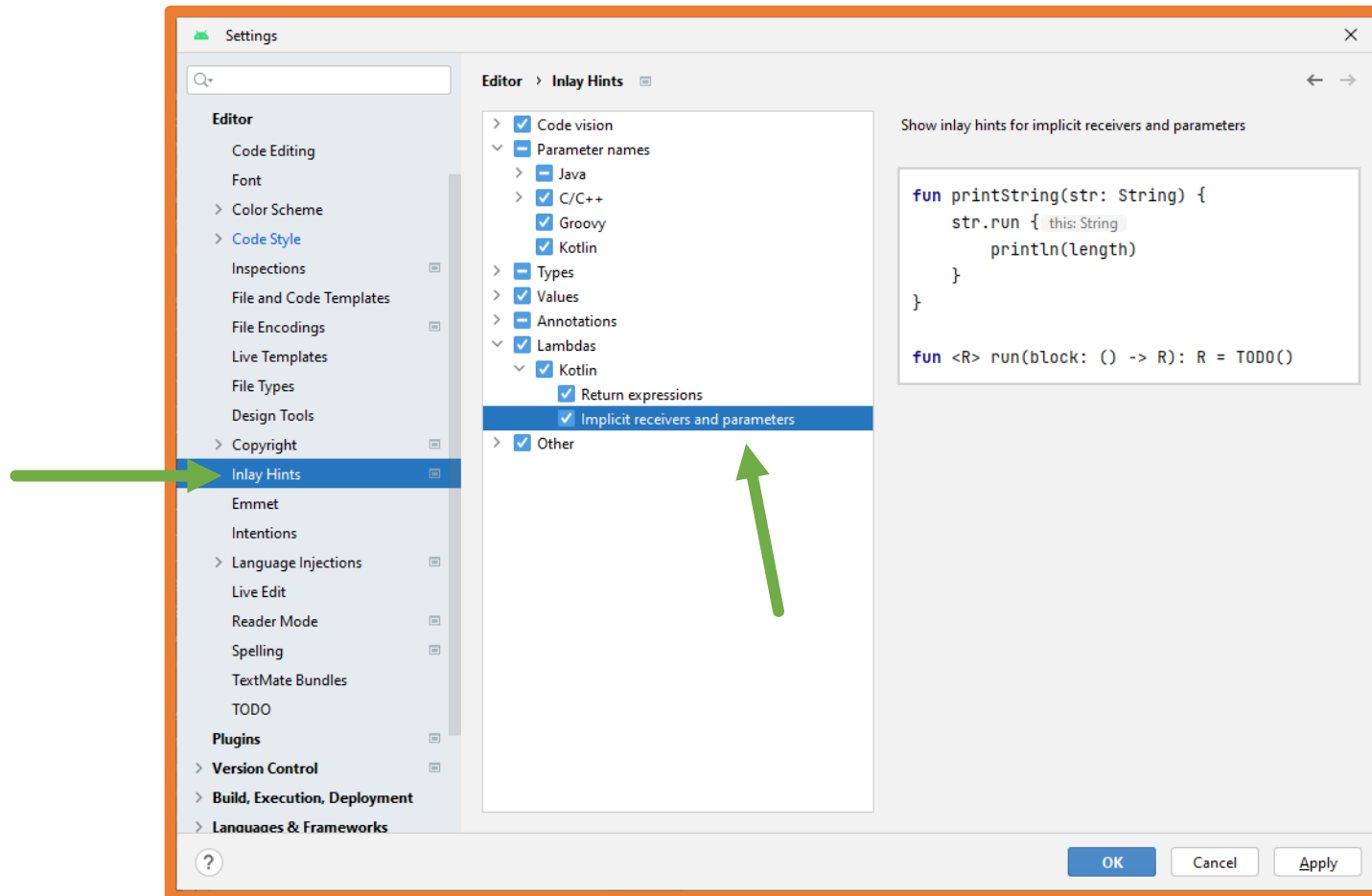


```
{  
    val first10EvenInts: MutableList<Int> = mutableListOf()  
    for (i in 0 until 10) {  
        first10EvenInts.add(i * 2)  
    }  
    first10EvenInts ^lambda  
}
```



7.- Funciones lambda

Si no aparecen las pistas de código (hints) en IntelliJ IDEA se pueden activar desde la configuración **File → Settings (CTRL+ALT+S)**:



7.- Funciones lambda

Si tiene parámetros, para separarlos del cuerpo se utilizan los caracteres -> de ahí que se les conozca también como funciones flecha.

```
{ name: String -> "Hola $name! Te saludo desde una lambda" }
```

```
{ x: Int, y: Int -> x + y }
```

```
{ init: Int, final: Int ->
    val evenInterval: MutableList<Int> = mutableListOf()
    for (i in init ≤ until < final) {
        if (i % 2 == 0) evenInterval.add(i)
    }
    evenInterval ^lambda
}
```


7.- Funciones lambda

En las lambdas escritas anteriormente es Kotlin quien deduce los tipos de datos.

Se pueden indicar explícitamente los tipos de datos:

```
(String) -> String = { name: String ->
    "Hola $name! Te saludo desde una lambda"
}
```

```
(Int, Int) -> MutableList<Int> = { init: Int, final: Int ->
    val evenInterval: MutableList<Int> = mutableListOf()
    for (i in init..until< final) {
        if (i % 2 == 0) evenInterval.add(i)
    }
    evenInterval ^lambda
}
```

En este caso si la lambda solo tiene un parámetro se puede omitir y utilizar **it**:

```
(String) -> String = { it: String
    "Hola $it! Te saludo desde una lambda"
}
```

7.- Funciones lambda

Las funciones lambda se utilizan como una expresión, esto significa que deben de estar asignadas a algún elemento.

Si se asigna una lambda a una variable su uso es el mismo que con una función normal.

Esto no aporta ninguna funcionalidad nueva.

```
val greeting = { name: String -> "Hola $name! Te saludo desde una lambda" }  
println(greeting("Rick"))
```

```
val add: (Int, Int) -> Int = { number1: Int, number2: Int -> number1 + number2 }  
println(add(3,48))
```

```
val evenInterval = { init: Int, final: Int ->  
    val evenInterval: MutableList<Int> = mutableListOf()  
    for (i in init until final) {  
        if (i % 2 == 0) evenInterval.add(i)  
    }  
    evenInterval ^lambda  
}  
println(evenInterval(0, 25))
```

7.- Funciones lambda

Las funciones lambda **se pueden pasar como parámetros a otras funciones**.

Esto sí que aporta una funcionalidad no vista hasta ahora y es una práctica muy extendida actualmente en muchos lenguajes de programación y frameworks.

Gracias al paso de lambdas como parámetro en funciones se puede:

- Crear funciones callback.
- Devolver diferentes respuestas desde una función.
- Ofrecer al programador que introduzca su propia lógica.

7.- Funciones lambda

Sintaxis de paso de función lambda como parámetro de una función:

```
fun nameOfFun(parameter1: Type, lambdaName: (Type1, Type2...) -> TypeReturn) { ... }
```

En el caso de que la lambda no reciba parámetros:

```
fun nameOfFun(parameter1: Type, lambdaName: () -> TypeReturn) { ... }
```

En el caso de que la lambda no devuelva nada:

```
fun nameOfFun(parameter1: Type, lambdaName: (Type1, Type2...) -> Unit) { ... }
```

En el caso de que la lambda no reciba parámetros ni devuelva nada:

```
fun nameOfFun(parameter1: Type, lambdaName: () -> Unit) { ... }
```

7.- Funciones lambda

Crear funciones callback

Las funciones callback permiten asegurarse que un conjunto de instrucciones se ejecuta después de una instrucción concreta. Su uso es muy típico cuando existe ejecución asíncrona de instrucciones, hilos...

Las siguientes funciones sobrecargadas **reciben una función lambda** que se ejecuta al final de todo el cuerpo de la función **doLogin**:

```
fun doLogin(username: String, pass: String, lambda: (String, String) -> Unit) {  
    // Instrucciones que conectan a la BBDD y se realiza el login  
    Thread.sleep(5000)  
    lambda(username, pass)  
}  
  
fun doLoginGuest(lambda: () -> Unit) {  
    // Instrucciones que conectan a la BBDD y se realiza el login invitado  
    Thread.sleep(5000)  
    lambda()  
}
```

7.- Funciones lambda

Crear funciones callback

Ejemplos de llamadas a la función **doLogin**.

```
fun doLogin(username: String, pass: String, lambda: (String, String) -> Unit) {  
    // Instrucciones que conectan a la BBDD y se realiza el login  
    Thread.sleep(5000)  
    lambda(username, pass)  
}  
  
fun doLoginGuest(lambda: () -> Unit) {  
    // Instrucciones que conectan a la BBDD y se realiza el login invitado  
    Thread.sleep(5000)  
    lambda()  
}
```

```
doLogin( username: "Rick", pass: "c-137", { username, pass -> println("Login: $username - $pass") })
```

Si la función lambda es el último parámetro se puede sacar fuera de los paréntesis:

```
doLogin( username: "Morty", pass: "Sm1th") { username, pass ->  
    println("Login: $username - $pass")  
}
```

Si la función lambda es el único parámetro se pueden quitar los paréntesis de la llamada:

```
doLoginGuest {  
    println("Login invitado")  
}
```

7.- Funciones lambda

Devolver diferentes respuestas desde una función

A una función se le pueden pasar tantos parámetros como se quiera y todos/varios de esos parámetros pueden ser una función lambda.

De esta manera se puede ejecutar una u otra función lambda según se necesite.

```
fun calculate(  
    number1: Double,  
    number2: Double,  
    operation: Char  
) : Double {  
    when (operation) {  
        '+' -> return number1 + number2  
        '-' -> return number1 - number2  
        '*' -> return number1 * number2  
        '/' -> return number1 / number2  
        else -> return 0.0  
    }  
}
```

```
fun calculate(  
    number1: Double,  
    number2: Double,  
    operation: Char,  
    correct: (result: Double) -> Unit,  
    error: (error: String) -> Unit  
) {  
    if (number1 < 0) {  
        error( error: "El primer número es menor que cero.")  
    } else if (number2 < 0) {  
        error( error: "El segundo número es menor que cero.")  
    } else {  
        when (operation) {  
            '+' -> correct( result: number1 + number2)  
            '-' -> correct( result: number1 - number2)  
            '*' -> correct( result: number1 * number2)  
            '/' -> {  
                if (number2 == 0.0) error( error: "No se puede dividir entre cero.")  
                else correct( result: number1 / number2)  
            }  
            else -> error( error: "Operación no válida")  
        }  
    }  
}
```

7.- Funciones lambda

Devolver diferentes respuestas desde una función

```
fun calculate(
    number1: Double,
    number2: Double,
    operation: Char,
    correct: (result: Double) -> Unit,
    error: (error: String) -> Unit
) {
    if (number1 < 0) {
        error( error: "El primer número es menor que cero.")
    } else if (number2 < 0) {
        error( error: "El segundo número es menor que cero.")
    } else {
        when (operation) {
            '+' -> correct( result: number1 + number2)
            '-' -> correct( result: number1 - number2)
            '*' -> correct( result: number1 * number2)
            '/' -> {
                if (number2 == 0.0) error( error: "No se puede dividir entre cero.")
                else correct( result: number1 / number2)
            }
            else -> error( error: "Operación no válida")
        }
    }
}
```

Uso:

```
var result = ""
calculate(
    number1 = 9.0,
    number2 = 6.0,
    operation = '-',
    correct = { it: Double
        result = it.toString()
    },
    error = { it: String
        result = it
    }
)
println("El resultado es: $result")
```

Salida:

El resultado es: 3.0

7.- Funciones lambda

Devolver diferentes respuestas desde una función

Dependiendo del cuerpo de la función lambda se podrán ejecutar una o varias de estas funciones.

En el ejemplo, tanto en la función **correct** como en la función **error** se guarda el resultado obtenido devuelto por lo que solo tendrá efecto última llamada a las funciones lambda sea cual sea:

```
fun calculate(
    number1: Double,
    number2: Double,
    operation: Char,
    correct: (result: Double) -> Unit,
    error: (error: String) -> Unit
) {
    if (number1 < 0) {
        error( error: "El primer número es menor que cero.")
    } else if (number2 < 0) {
        error( error: "El segundo número es menor que cero.")
    } else {
        error( error: "Sin error")
        correct( result: 0.0)
        when (operation) {
            '+' -> correct( result: number1 + number2)
            '-' -> correct( result: number1 - number2)
            '*' -> correct( result: number1 * number2)
            '/' -> {
                if (number2 == 0.0) error( error: "No se puede dividir entre cero.")
                else correct( result: number1 / number2)
            }
            else -> error( error: "Operación no válida")
        }
    }
}
```

Uso:

```
var result = ""
calculate(
    number1 = 9.0,
    number2 = 6.0,
    operation = '-',
    correct = { it: Double
        result = it.toString()
    },
    error = { it: String
        result = it
    }
)
println("El resultado es: $result")
```

Salida:

El resultado es: 3.0

7.- Funciones lambda

Devolver diferentes respuestas desde una función

Dependiendo del cuerpo de la función lambda se podrán ejecutar una o varias de estas funciones.

Ahora se ha cambiado el cuerpo de las funciones **correct** y **error** para que muestren el resultado obtenido:

```
fun calculate(
    number1: Double,
    number2: Double,
    operation: Char,
    correct: (result: Double) -> Unit,
    error: (error: String) -> Unit
) {
    if (number1 < 0) {
        error( error: "El primer número es menor que cero.")
    } else if (number2 < 0) {
        error( error: "El segundo número es menor que cero.")
    } else {
        error( error: "Sin error")
        correct( result: 0.0)
        when (operation) {
            '+' -> correct( result: number1 + number2)
            '-' -> correct( result: number1 - number2)
            '*' -> correct( result: number1 * number2)
            '/' -> {
                if (number2 == 0.0) error( error: "No se puede dividir entre cero.")
                else correct( result: number1 / number2)
            }
            else -> error( error: "Operación no válida")
        }
    }
}
```

Uso:

```
calculate(
    number1 = 9.0,
    number2 = 6.0,
    operation = '-',
    correct = { it: Double
        println("El resultado es: $it")
    },
    error = { it: String
        println("Error: $it")
    }
)
```

Salida:

```
Error: Sin error
El resultado es: 0.0
El resultado es: 3.0
```

7.- Funciones lambda

Devolver diferentes respuestas desde una función

Si se quiere **definir un parámetro tipo función lambda como opcional** se debe indicar con los caracteres = {}.

```
fun calculate(  
    number1: Double,  
    number2: Double,  
    operation: Char,  
    correct: (result: Double) -> Unit,  
    error: (error: String) -> Unit = {}  
) {  
    if (number1 < 0) {  
        error( error: "El primer número es menor que cero.")  
    } else if (number2 < 0) {  
        error( error: "El segundo número es menor que cero.")  
    } else {  
        when (operation) {  
            '+' -> correct( result: number1 + number2)  
            '-' -> correct( result: number1 - number2)  
            '*' -> correct( result: number1 * number2)  
            '/' -> {  
                if (number2 == 0.0) error( error: "No se puede dividir entre cero.")  
                else correct( result: number1 / number2)  
            }  
        }  
        else -> error( error: "Operación no válida")  
    }  
}
```

Uso:

```
calculate(  
    number1 = 9.0,  
    number2 = 6.0,  
    operation = 'o',  
    correct = { it: Double  
        println("El resultado es: $it")  
    },  
    error = { it: String  
        println("Error: $it")  
    }  
)
```

Salida:

Error: Operación no válida

Uso:

```
calculate(  
    number1 = 9.0,  
    number2 = 6.0,  
    operation = 'o',  
    correct = { it: Double  
        println("El resultado es: $it")  
    },  
)
```

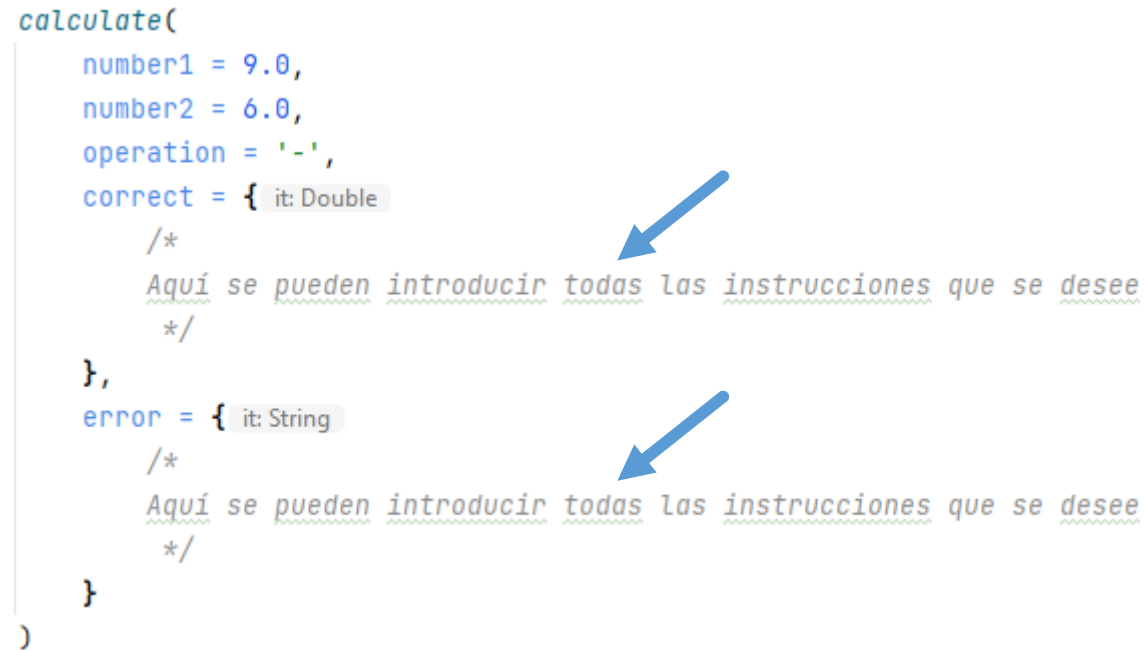
Salida:

7.- Funciones lambda

Ofrecer al programador que introduzca su propia lógica

Si la función **calculate** estuviera en una librería externa al usarla se da la opción al programador de incorporar todas las instrucciones que quiera.

```
calculate(  
    number1 = 9.0,  
    number2 = 6.0,  
    operation = '-',  
    correct = { it: Double  
        /*  
        Aquí se pueden introducir todas las instrucciones que se desee  
        */  
    },  
    error = { it: String  
        /*  
        Aquí se pueden introducir todas las instrucciones que se desee  
        */  
    }  
)
```


The diagram shows a code snippet for a function named 'calculate'. The function has four parameters: 'number1' (9.0), 'number2' (6.0), 'operation' ('-'), and a block parameter 'correct' of type 'Double'. The 'correct' block contains a comment indicating where instructions can be added. A blue arrow points from the text 'Aquí se pueden introducir todas las instrucciones que se desee' to the 'correct' block. Another blue arrow points from the same text to the 'error' block, which is also of type 'String' and contains a similar comment. The function ends with a closing parenthesis ')

7.- Funciones lambda

Anteriormente ya se ha hecho uso de funciones lambda como parámetros de otras funciones, exactamente como último parámetro de la función:


- En la creación de arrays:

```
var reverse = Array( size: 50) { 50-it }
```




- En las funciones forEach, filter y map:

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



- En las funciones de alcance:

```
val product: Product = Product( name: "Nintendo Switch", price: 300.0, year: 2021)
product.apply { this: Product
    // this.name = "Switch OLED"
    name = "Switch OLED"
    price = 349.90
}
```



7.- Funciones lambda

El uso de funciones lambda es muy importante en el desarrollo de aplicaciones móviles Android.

Sobre todo al usar **Jetpack Compose** como se verá en el tema siguiente.

```
MyMoviesApp {  
    Scaffold(topBar = { MainAppBar(mediaItem.title) }) { it: PaddingValues  
        Row(  
            modifier = Modifier.padding()  
        ) { this: RowScope  
            MainContent(  
                onNavigate = { it: MediaItem  
                    currentItem = it.id  
                },  
                modifier = Modifier.weight(1f)  
            )  
            DetailContent(  
                mediaItem = mediaItem,  
                modifier = Modifier.weight(2f)  
            )  
        }  
    }  
}
```

Lambdas



Práctica

Actividad 6

Electrodomésticos