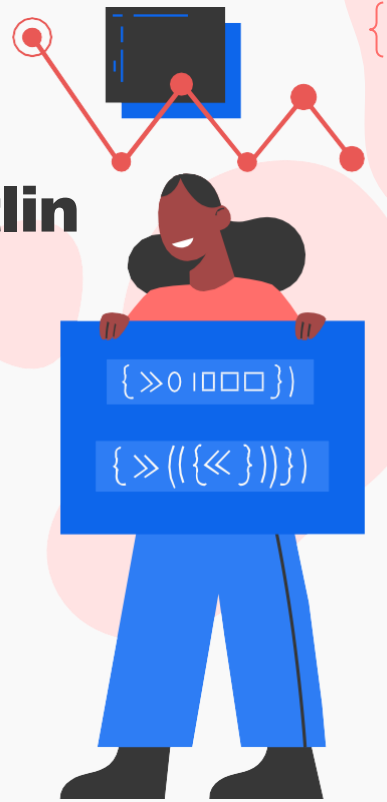
Fundamentos de kotlin

Tabla de contenidos



01

Introducción

¿Qué es?

Kotlin es un lenguaje de programación de propósito general, estáticamente tipado, que se ejecuta en la Máquina Virtual de Java (JVM) y puede compilar a código JavaScript o código nativo.

Caracteristicas

- Interoperabilidad con Java: Puede trabajar junto con código Java
- Sintaxis concisa: Reduce la cantidad de código boilerplate comparado con Java, lo que mejora la legibilidad.
- Soporte para programación funcional: Admite funciones de orden superior y expresiones lambda.

Historia

- Kotlin fue desarrollado por JetBrains y presentado por primera vez en 2011.
- La versión 1.0 fue lanzada en febrero de 2016.
- En 2017, Google anunció Kotlin como un lenguaje oficial para el desarrollo de Android.

Ventajas

Concisión

Permite escribir menos código para lograr el mismo resultado, lo que mejora la productividad.

Seguridad

Incluye características de seguridad, como un sistema de tipos que ayuda a prevenir errores comunes, como las excepciones de puntero nulo (NullPointerException).

Ventajas

Modernidad

Ofrece características modernas como corutinas para programación asíncrona.

Ecosistema

Se beneficia del amplio ecosistema de Java y su biblioteca estándar.

Interoperabilidad

Se integra perfectamente con proyectos Java existentes. Puedes utilizar bibliotecas y frameworks de Java.

02

Sintaxis,

Variables, tipos de datos y estructuras de control

Declaración de variables

Variables en Kotlin: Kotlin ofrece dos palabras clave principales para declarar variables:

- `val`: Define una variable de solo lectura, similar a una constante. Su valor no puede ser cambiado una vez asignado.
- `var`: Define una variable mutable. Su valor puede ser reasignado después de la declaración.

```kotlin
// Declaración de variables en Kotlin
val nombre: String = "Juan" // Variable inmutable
var edad: Int = 25 // Variable mutable


//Inferencia de variables
val mensaje = "Hola, soy una variable inferida"

// Imprimir variables
println("Nombre: $nombre")
println("Edad: $edad")
println("inferida: $mensaje")
```

Tipos de
Datos y operadores

Tipos de datos básicos:

- **Números**: Int, Long, Float, Double, Short, Byte.
- **Texto**: String y Char.
- **Booleano**: Boolean (true o false).

Operadores en Kotlin:

- **Aritméticos**: `+`, `-`, `*`, `/`, `%`.
- **Comparación**: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- **Lógicos**: `&&`, `||`, `!`.
- **Asignación**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`.

Estructuras de control

La estructura **if** funciona de forma similar que en otros lenguajes, sin embargo, en kotlin es capaz de retornar un valor.

```kotlin

val numero = 5

val resultado = if (numero % 2 == 0) "Par" else "Impar"

println("El número es $resultado") // Imprime: "El número es Impar"

```

Estructura **for**

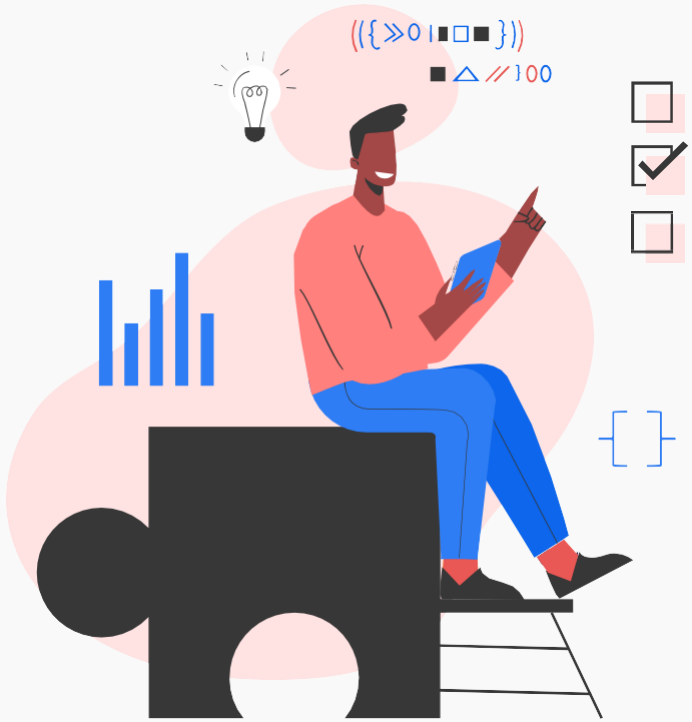
```kotlin
for (i in 1..5) {
    println("Número: $i") // Imprime del 1 al 5
}
for (i in 5 downTo 1) {
    println("Número: $i") // Imprime del 5 al 1
}
for (i in 1..10 step 2) {
    println("Número: $i") // Imprime 1, 3, 5, 7, 9
}
```

Estructura **while**

```kotlin
var contador = 3
while (contador > 0) {
    println("Contando: $contador")
    contador-- // Decrementa el valor de contador
}
```

03

POO en Kotlin

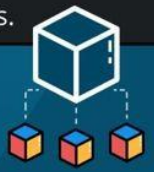


¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS?

Es un paradigma de programación que organiza las funciones en entidades llamadas objetos.

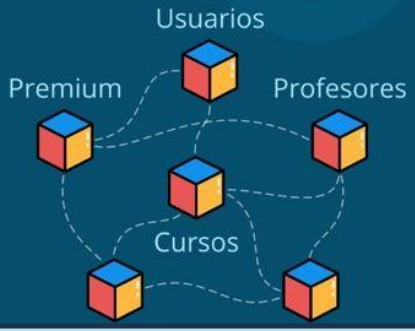
- **Los objetos se crean a partir de una plantilla llamada clase**. Cada objeto es una instancia de su clase.

CLASE → INSTANCIACIÓN → OBJETO



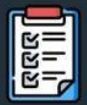
- Los objetos tienen **datos (atributos)** y **funcionalidades (métodos).**

- En una aplicación los objetos están separados **pero se comunican entre ellos.**



ATRIBUTOS

Nombres
Apellidos
Correo
Contraseña
Premium

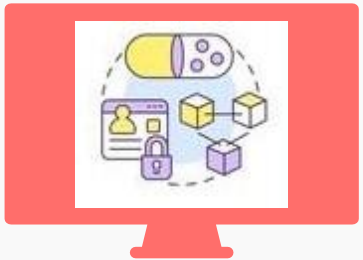


MÉTODOS

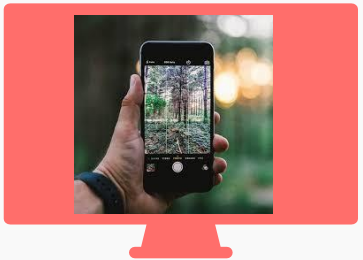
Editar perfil
Iniciar sesión
Cerrar sesión
Cambiar contraseña
Pasar a premium

Puedes programar con este paradigma **en la mayoría de lenguajes.**

Conceptos de POO

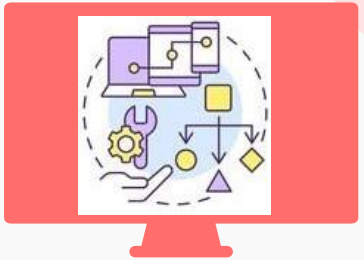


Encapsulamiento



Abstracción

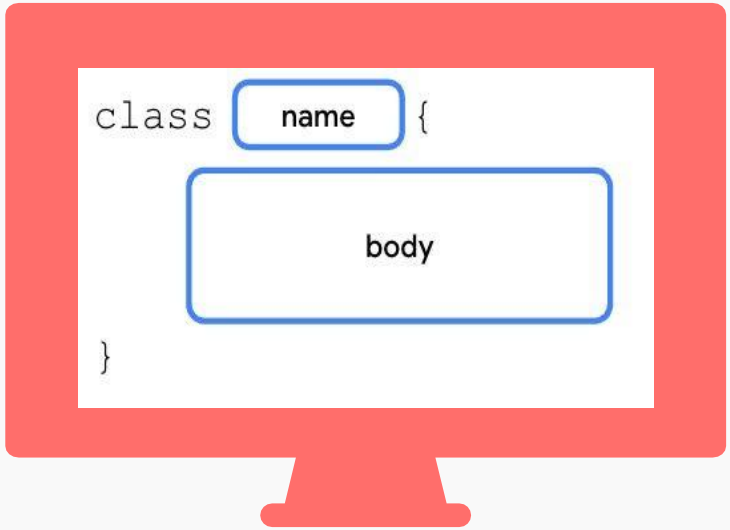
Herencia



Polimorfismo

Clases y Objetos en Kotlin

```
class SmartDevice {
    // empty body
}

fun main() {
}
```



Instancia en Kotlin


```
fun main() {
    val smartTvDevice = SmartDevice()
}
```

Métodos en Kotlin

Cuando defines una *función* en el cuerpo de la clase, se la conoce como una función de miembro o *método*, y representa el comportamiento de la clase.

```
class SmartDevice {
    fun turnOn() {

    }
}
```

```
class SmartDevice {
    fun turnOn() {
        println("Smart device is turned on.")
    }

    fun turnOff() {
        println("Smart device is turned off.")
    }
}
```

Llama a un método en un objeto.


```
fun main() {
    val smartTvDevice = SmartDevice()
    smartTvDevice.turnOn()
    smartTvDevice.turnOff()
}
```

Propiedades de las clases.

Mientras que los métodos definen las acciones que puede realizar una clase, las propiedades definen las características o los atributos de los datos de la clase.

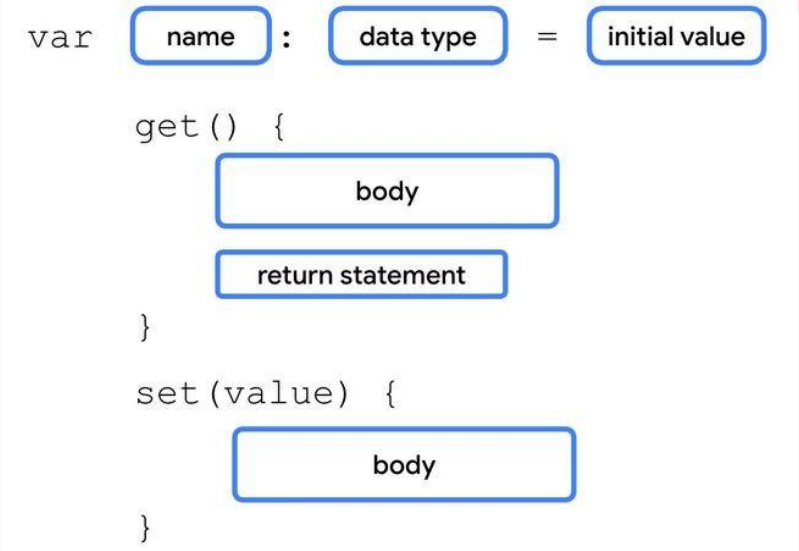
```
class SmartDevice {

    val name = "Android TV"
    val category = "Entertainment"
    var deviceStatus = "online"

    fun turnOn() {
        println("Smart device is turned on.")
    }

    fun turnOff() {
        println("Smart device is turned off.")
    }
}
```

```
fun main() {
    val smartTvDevice = SmartDevice()
    println("Device name is: ${smartTvDevice.name}")
    smartTvDevice.turnOn()
    smartTvDevice.turnOff()
}
```

Funciones get y set

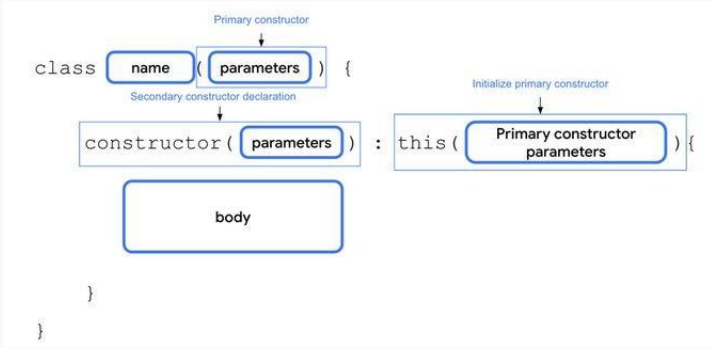
Cuando no defines la función de método get y set para una propiedad, el compilador de Kotlin crea las funciones a nivel interno.

```
var speakerVolume = 2
    get() = field
    set(value) {
        field = value
    }
```

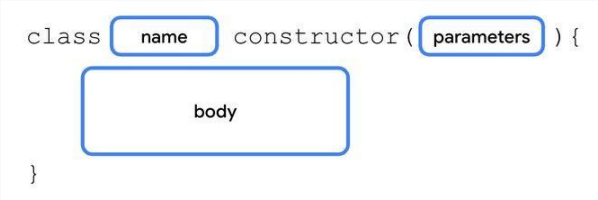


Constructores en Kotlin.

El objetivo principal del *constructor* es especificar cómo se crean los objetos de la clase. En otras palabras, los constructores inicializan un objeto y lo preparan para su uso.



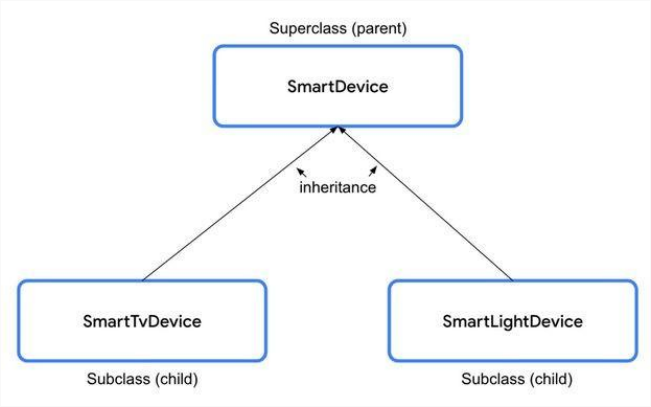

```
class SmartDevice constructor() {
    ...
}
```




```
class SmartDevice(val name: String, val category: String) {
    var deviceStatus = "online"

    constructor(name: String, category: String, statusCode: Int) : this(name, category)
        deviceStatus = when (statusCode) {
            0 -> "offline"
            1 -> "online"
            else -> "unknown"
        }
    }
    ...
}
```

Herencia en Kotlin.

La herencia te permite compilar una clase en función de las características y el comportamiento de otra clase.

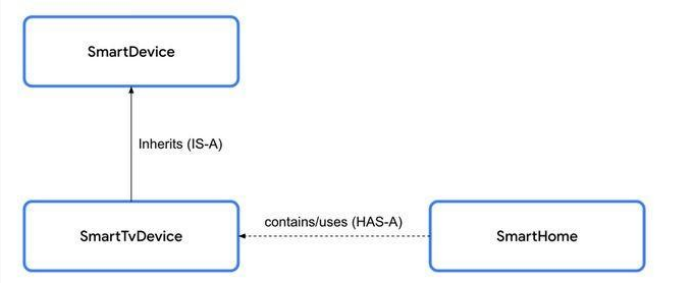



```
open class SmartDevice(val name: String, val category: String) {
    ...
}
```

```
class Subclass name ( [optional] parameters ) :
      Superclass name ( [optional] parameters ) {

      body
}
```

Relaciones entre clases

Una relación entre dos clases en *relación IS-A*. Un objeto también es una instancia de la clase de la cual se hereda.
En una *relación HAS-A*, un objeto puede tener una instancia de otra clase sin ser realmente una instancia de esa clase en sí.




```
// Smart TV IS-A smart device.
class SmartTvDevice : SmartDevice() {
}


// The SmartHome class HAS-A smart TV device.
class SmartHome(val smartTvDevice: SmartTvDevice) {

}
```

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    ...

}

class SmartHome {
}

fun main() {
    ...
}
```

Anulación de métodos

La anulación de un método implica interceptar la acción, generalmente para realizar un control manual.
Cuando anulas un método, el método de la subclase interrumpe la ejecución del método definido en la superclase y proporciona su propia ejecución.

```
open class SmartDevice(val name: String, val category: String) {

    var deviceStatus = "online"

    open fun turnOn() {
        // function body
    }

    open fun turnOff() {
        // function body
    }
}
```

```
class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var speakerVolume = 2
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }

    var channelNumber = 1
        set(value) {
            if (value in 0..200) {
                field = value
            }
        }

    fun increaseSpeakerVolume() {
        speakerVolume++
        println("Speaker volume increased to $speakerVolume.")
    }

    fun nextChannel() {
        channelNumber++
        println("Channel number increased to $channelNumber.")
    }

    override fun turnOn() {
        deviceStatus = "on"
        println(
            "$name is turned on. Speaker volume is set to $speakerVolume and channel num
                "set to $channelNumber."
        )
    }

    override fun turnOff() {
        deviceStatus = "off"
        println("$name turned off")
    }
}
```

`super.` **functionName** `(` **[Optional] Arguments** `)`

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    var brightnessLevel = 0
        set(value) {
            if (value in 0..100) {
                field = value
            }
        }

    fun increaseBrightness() {
        brightnessLevel++
        println("Brightness increased to $brightnessLevel.")
    }

    override fun turnOn() {
        super.turnOn()
        brightnessLevel = 2
        println("$name turned on. The brightness level is $brightnessLevel.")
    }

    override fun turnOff() {
        super.turnOff()
        brightnessLevel = 0
        println("Smart Light turned off")
    }
}
```

```
open class SmartDevice(val name: String, val category: String) {

    var deviceStatus = "online"

    open val deviceType = "unknown"
    ...
}
```

```
class SmartLightDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {

    override val deviceType = "Smart Light"

    ...

}
```

Modificador	Accesible en la misma clase	Accesible en subclase	Accesible en el mismo módulo	Accesible fuera del módulo
private	✓	X	X	X
protected	✓	✓	X	X
internal	✓	✓	✓	X
public	✓	✓	✓	✓

04

Funciones y Lambdas

Declaración y llamada de Funciones

Declaración de Funciones

En Kotlin, las funciones se declaran utilizando la palabra clave `fun`. La sintaxis básica es:

```
fun saludar(nombre: String): String {
    return "Hola, $nombre!"
}
```

- **`fun`**: palabra clave para declarar una función.
- **`nombreDeLaFuncion`**: nombre descriptivo de la función.
- **`param1: Tipo1, param2: Tipo2`**: parámetros de la función con sus tipos.
- **`: TipoDeRetorno`**: tipo de dato que retorna la función (puede omitirse si es `Unit`).
- **`return valor`**: retorna un valor de tipo `TipoDeRetorno`.

Llamada de Funciones

Para llamar a una función, simplemente se usa su nombre seguido de los argumentos entre paréntesis.

```
fun main() {
    val mensaje = saludar( nombre: "Juan")
    println(mensaje) // Imprime: Hola, Juan!
}
```

Funciones con valores por defecto y parámetros nombrados

Kotlin permite definir valores por defecto para los parámetros y también usar parámetros nombrados al llamar a la función.

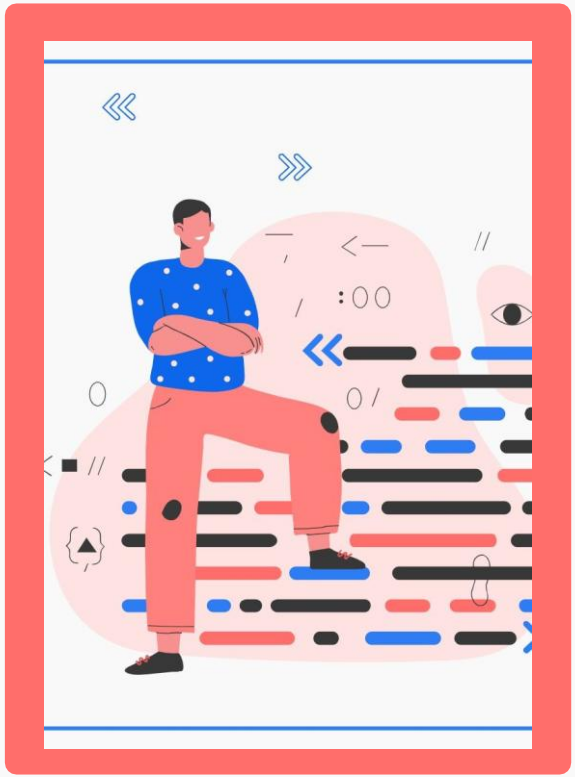
```kotlin
fun saludar(nombre: String, saludo: String = "Hola"): String {
    return "$saludo, $nombre!"
}

fun main() {
    println(saludar( nombre: "Ana")) // Usa el valor por defecto: Hola, Ana!
    println(saludar( nombre: "Ana",  saludo: "Buenos días")) // Personaliza el saludo: Buenos días, Ana!
    println(saludar(saludo = "Saludos", nombre = "Ana")) // Uso de parámetros nombrados
}
```

Funciones de Orden Superior

Las funciones de orden superior son aquellas que pueden recibir otras funciones como parámetros o devolver funciones. Esto permite una programación más abstracta y reutilizable.

Ejemplo de uso

```
fun calcular(n1: Int, n2: Int, operacion: (Int, Int) -> Int): Int {
    return operacion(n1, n2)
}

fun sumar(x: Int, y: Int) = x + y

fun restar(x: Int, y: Int) = x - y

fun multiplicar(x: Int, y: Int) = x * y

fun dividir(x: Int, y: Int) = x / y

fun main(parametro: Array<String>) {
    val resultado = calcular( n1: 20,  n2: 5, ::sumar)
    println("20 + 5 = $resultado") // 20 + 5 = 25
    println("10 - 4 = ${calcular( n1: 10,  n2: 4, ::restar)}") // 10 - 4 = 6
    println("12 * 2 = ${calcular( n1: 12,  n2: 2, ::multiplicar)}") // 12 * 2 = 24
    println("50 / 2 = ${calcular( n1: 50,  n2: 2, ::dividir)}") // 50 / 2 = 25
}
```



Lambdas y Funciones anónimas

Funciones Anónimas

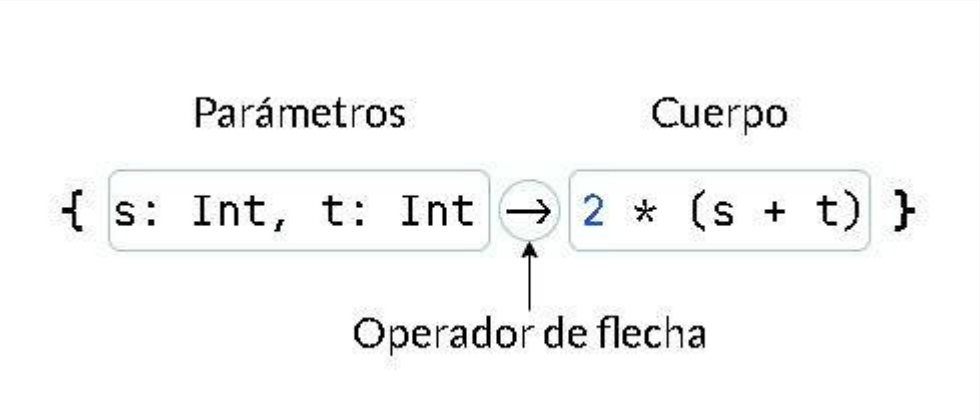
Las funciones anónimas (y también las expresiones lambda) son 'funciones literales', es decir, funciones que no se declaran pero se pasan de inmediato como una expresión.

```
fun main(args: Array<String>) {
    val multiplicarPorCinco = fun(a: Int) = a * 5 // función anónima
    println(multiplicarPorCinco(4)) // 20
}
```

Normalmente se crean y se utilizan en el mismo lugar.

Lambda

Una lambda es una función anónima que se utiliza en el mismo momento que se declara. Su sintaxis básica es:



- **`Parametros`**: Cada parámetro es una declaración de variable, aunque esta lista es opcional
- **`Operador flecha(->)`**: Se omite si no usas lista de parámetros.
- **`Cuerpo`**: Son las sentencias que van luego del operador de flecha

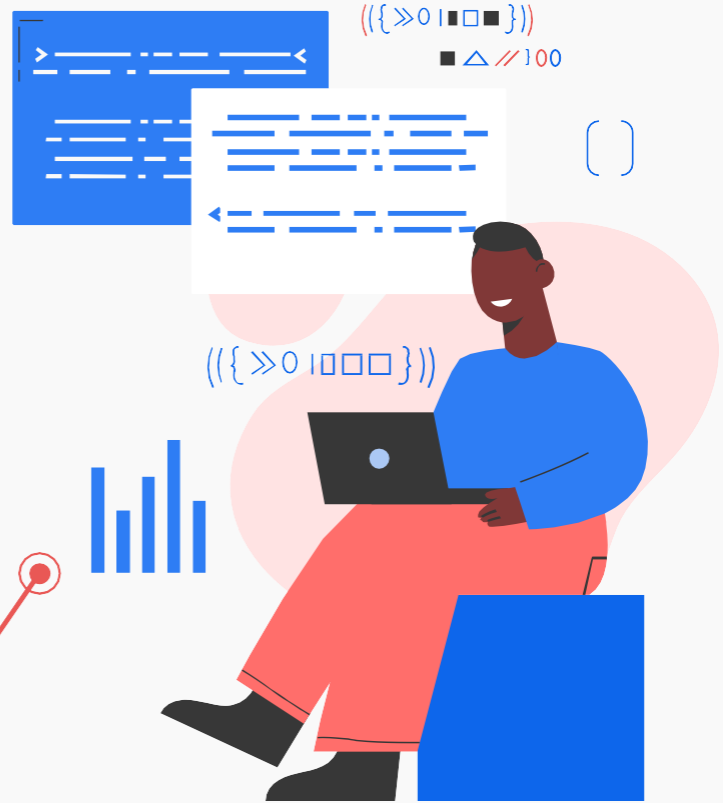
Ejemplo de uso

```
fun transformar(n: Int, anonima: (Int) -> Int) = anonima(n)

fun main() {
    val resultado = transformar( n: 2, { a: Int -> a * 5 }) // lambda que multiplica por 5
    println(resultado) // Salida: 10
}
```

No es necesario escribir `fun` ni `return`. El valor que devuelve la lambda es la última expresión dentro de las llaves `{}`.

`transformar()` es una función que toma un `Int` (`n`) y una función que a su vez recibe un `Int` y devuelve un `Int` (el parámetro `anonima`).
En `transformar(2, { a: Int -> a * 5 })`, la lambda `{ a: Int -> a * 5 }` es una función que toma `a` y devuelve `a * 5`. Esto significa que `transformar(2, { a: Int -> a * 5 })` aplicará esta función lambda al valor `2`, resultando en `10`.

Manejo de excepciones y concurrencia



Excepciones en Kotlin

Las excepciones ayudan a que el código se ejecute de forma más predecible, incluso cuando se producen errores de ejecución que podrían interrumpir la ejecución del programa. Kotlin trata todas las excepciones como no controladas de forma predeterminada.

Trabajar con excepciones consta de dos acciones principales:

- Throwing exceptions: Indica cuándo ocurre un problema.
- Catching exceptions: Gestiona la excepción inesperada manualmente resolviendo el problema o notificando al desarrollador o al usuario de la aplicación.

Throwing exceptions

```
throw IllegalArgumentException()
```

```
val cause = IllegalStateException("Original cause: illegal state")

// Throws an IllegalArgumentException if userInput is negative
// Additionally, it shows the original cause, represented by the cause IllegalStat
if (userInput < 0) {
    throw IllegalArgumentException("Input must be non-negative", cause)
}
```

Throw exceptions with precondition functions

Función require()

Utilice la función require() para validar los argumentos de entrada cuando sean cruciales para el funcionamiento de la función y la función no puede continuar si estos argumentos no son válidos.

```
.versión  getIndices ( count : Int ): Lista < Int > {
  require ( count  >=  0 ) { "Count debe ser no negativo. Establece count en $co
  devuelve  Lista ( count ) { it  +  1 }


.versión  principal () {
  // Esto falla con una IllegalArgumentException
  println ( obtenerIndices ( - 1 ))

  // Descomente la línea a continuación para ver un ejemplo funcional
  // println(obtenerIndices(3))
  // [1, 2, 3]
```

```
Excepción en el hilo "principal" java.lang.IllegalArgumentException : El recuento no
debe ser negativo. Estableció el recuento en -1.
   en FileKt . getIndices ( File.kt : 2 )
   en FileKt . main ( File.kt : 8 )
   en FileKt . main ( File.kt : -1 )
```

Abierto en el patio de recreo → Objetivo: JVM Ejecutándose en la versión 2.0.21

Throw exceptions with precondition functions

Función check()

Utilice la función check() para validar el estado de un objeto o una variable. Si la comprobación falla, indica un error lógico que debe solucionarse.

```
fun main() {
    var someState: String? = null

    fun getStateValue(): String {

        val state = checkNotNull(someState) { "State must be set beforehand!" }
        check(state.isNotEmpty()) { "State must be non-empty!" }
        return state
    }
    // If you uncomment the line below then the program fails with IllegalStateE
    // getStateValue()

    someState = ""

    // If you uncomment the line below then the program fails with IllegalStateE
    // getStateValue()
    someState = ""

    // This prints "non-empty-state"
    println(getStateValue())
}
```

```
Exception in thread "main" java.lang.IllegalStateException: State must be non-empty!
    at FileKt.main$getStateValue (File.kt:7)
    at FileKt.main (File.kt:20)
    at FileKt.main (File.kt:-1)
```

Throw exceptions with precondition functions

Función error()

Utilice la función error() se utiliza para señalar un estado ilegal o una condición en el código que lógicamente no debería ocurrir.

```
class User(val name: String, val role: String)

fun processUserRole(user: User) {
    when (user.role) {
        "admin" -> println("${user.name} is an admin.")
        "editor" -> println("${user.name} is an editor.")
        "viewer" -> println("${user.name} is a viewer.")
        else -> error("Undefined role: ${user.role}")
    }
}

fun main() {
    // This works as expected
    val user1 = User("Alice", "admin")
    processUserRole(user1)
    // Alice is an admin.

    // This throws an IllegalStateException
    val user2 = User("Bob", "guest")
    processUserRole(user2)
}
```

```
Alice is an admin.
Exception in thread "main" java.lang.IllegalStateException: Undefined role: guest
    at FileKt.processUserRole (File.kt:8)
    at FileKt.main (File.kt:20)
    at FileKt.main (File.kt:-1)
```

Manejar excepciones usando bloques try-catch

```
fun main() {
    val num: Int = try {

        // If count() completes successfully, its return value is assigned to nu
        count()

    } catch (e: ArithmeticException) {

        // If count() throws an exception, the catch block returns -1,
        // which is assigned to num
        -1
    }
    println("Result: $num")
}

// Simulates a function that might throw ArithmeticException
fun count(): Int {

    // Change this value to return a different value to num
    val a = 0

    return 10 / a
}
```

```
Result: -1
```

Open in Playground → Target: JVM Running on v.2.0.21

Concurrencia con Coroutines

Las coroutines son una forma de manejar la concurrencia de forma ligera en Kotlin, permitiendo realizar tareas en segundo plano sin bloquear el hilo principal.
Son útiles para realizar tareas largas, como consultas a bases de datos o llamadas de red, sin congelar la interfaz de usuario.

Dispatchers:

- Dispatchers.Main
- Dispatchers.IO
- Dispatchers.Default

Scopes:

- GlobalScope
- runBlocking
- CoroutineScope

Concurrencia con Coroutines

```
class MiClase {
    private val scope = CoroutineScope(Dispatchers.Default)

    fun iniciarTarea() {
        scope.launch {
            // Tarea en segundo plano
            println("Coroutine en ejecución")
        }
    }

    fun detenerTareas() {
        scope.cancel() // Cancela todas las coroutines en este scope
    }
}
```

Manejo de hilos (Threads)

Un hilo es una unidad de ejecución dentro de un proceso. Los programas pueden tener múltiples hilos que se ejecutan de forma concurrente.

- Puedes crear hilos utilizando Thread o con la clase Runnable.

```
fun main() {
    val thread = Thread {
        println("Este es un hilo nuevo")
    }
    thread.start()
}
```

Manejo de hilos (Threads)

```
var contador = 0

@Synchronized
fun incrementarContador() {
    contador++
}

fun main() {
    val hilo1 = Thread { incrementarContador() }
    val hilo2 = Thread { incrementarContador() }
    hilo1.start()
    hilo2.start()
    hilo1.join()
    hilo2.join()
    println("Contador final: $contador")
}
```

Referencias

3“Usa clases y objetos en Kotlin | Android Developers”. Android Developers. Accedido el 7 de noviembre de 2024. [En línea]. Disponible: https://developer.android.com/codelabs/basic-android-kotlin-compose-classes-and-objects?hl=es-419#0

4 Android Developers, "Coroutines en Kotlin," Android Developers. Accedido el 7 de noviembre de 2024. [En linea]. Disponible: https://developer.android.com/kotlin/coroutines?hl=es-419

5 Android Developers, "Coroutines en Kotlin," Android Developers, https://developer.android.com/kotlin/coroutines?hl=es-419 (accedido el 7 de noviembre de 2024).

6 Android Developers, "Coroutines avanzadas en Kotlin," Android Developers, https://developer.android.com/kotlin/coroutines/coroutines-adv?hl=es-419 (accedido el 7 de noviembre de 2024).