Kotlin

Corrutinas

Una *corrutina* es un patrón de diseño de simultaneidad que puedes usar en Android para simplificar el código que se ejecuta de forma asíncrona.

Las corrutinas se agregaron a Kotlin en la versión 1.3 y se basan en conceptos establecidos de otros lenguajes.

Programación asíncrona

- Una línea de código se ejecuta tras terminar la anterior.
- Capacidad multihilo de lenguajes de programación, como Java para ejecutar tareas de forma asíncrona.
- Se puede crear un hilo de trabajo separado para realizar ciertas tareas : descarga de ficheros, peticiones al servidor, consultas a la base de datos…
- El hilo principal (la interfaz gráfica) no se bloquea y la aplicación se muestra fluida.
- Kotlin propone las corrutinas como solución a la programación asíncrona.

Corrutinas en Kotlin

- Una corrutina es un conjunto de sentencias que realizan una tarea específica, con la capacidad suspender o resumir su ejecución sin bloquear un hilo.
- Esto permite que tengas diferentes corrutinas cooperando entre ellas, suspendiéndose y resumiéndose en puntos especificados por ti o por Kotlin.
- La diferencia con los hilos es que no significa que exista un hilo por cada corrutina, al contrario, puedes ejecutar varias en un solo.
  - Procesamiento concurrente.

Corrutinas en Kotlin

- Reducir recursos del sistema al evitar la creación de grandes cantidades de hilos
- Facilitar el retorno de datos de una tarea asíncrona
- Facilitar el intercambio de datos entre tareas asíncronas

En Android, las corrutinas ayudan a administrar tareas de larga duración que, de lo contrario, podrían bloquear el subproceso principal y hacer que tu app dejara de responder.

Corrutines en InteliiJ

- Las corrutinas se encuentran en el paquete kotlinx.coroutines.
- Se debe especificar la dependencia en la configuración de build.gradle.kts
- Luego añade al bloque repositories()
- No olvidemos sincronizar el proyecto.
- Una vez se complete la sincronización, ya puedes iniciar corrutinas.

```
dependencies { /* Otras dependencias
*/
implementation("org.jetbrains.kotlinx:
kotlinx-coroutines-core:1.4.2") }

repositories { jcenter() }
```

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9")
}
```

Para usar corrutinas en tu proyecto de Android, agrega la siguiente dependencia al archivo build.gradle de tu app

```
sealed class Result<out R> {
    data class Success<out T>(val data: T) : Result<T>()
    data class Error(val exception: Exception) : Result<Nothing>()
}

class LoginRepository(private val responseParser: LoginResponseParser) {
    private const val loginUrl = "https://example.com/login"

    // Function that makes the network request, blocking the current thread
    fun makeLoginRequest(
        jsonBody: String
    ): Result<LoginResponse> {
        val url = URL(loginUrl)
        (url.openConnection() as? HttpURLConnection)?.run {
            requestMethod = "POST"
            setRequestProperty("Content-Type", "application/json; utf-8")
            setRequestProperty("Accept", "application/json")
            doOutput = true
            outputStream.write(jsonBody.toByteArray())
            return Result.Success(responseParser.parse(inputStream))
        }
        return Result.Error(Exception("Cannot open HttpURLConnection"))
    }
}
```

Iniciar las corrutinas

- Para iniciar una corrutina debes usar un constructor de corrutinas (launch(), runBlocking(), async(), etc.) y pasar un lambda con las sentencias a ejecutar como bloque de código.

```
// Importar componentes de corrutinas
import kotlinx.coroutines.*

fun main() {
    // 1. Inicio
    println("¡Go!")

    // 2. Buscar palabras en el background
    GlobalScope.launch {
        (1..5).forEach {
            delay(300)
            println("¡Palabra $it encontrada!")
        }
    }

    // 3. Iniciar temporizador en foreground
    for (i in 10 downTo 1) {
        println("${i}s")
        Thread.sleep(100)
    }

    // 4. Tiempo fuera
    println("Se terminó el tiempo")
}
```

Este programa marca las palabras encontradas por un usuario en una sopa de letras.
Tiene 10 segundos de tiempo.

Solución

- Iniciar la corrutina con GlobalScope.launch{}.
- Esta cuenta las palabras encontradas.
- Usamos la función de suspención delay() para simular que el usuario se demora 300 milisegundos encontrando cada palabra
- Iniciar un bucle for en reversa para simular el temporizador de 10 a 1.
- Usamos el método Thread.sleep() de la librería estándar de Java, para dormir el hilo principal (main)

- Al ejecutar la aplicación se imprimirá:
  - La esperanza de vida de la corrutina se extendió solo hasta que terminó el último println().
  - Por eso faltaron 2 palabras por encontrar.

```
¡Go!
10s
9s
8s
7s
¡Palabra 1 encontrada!
6s
5s
4s
¡Palabra 2 encontrada!
3s
2s
1s
¡Palabra 3 encontrada!
Se terminó el tiempo
```

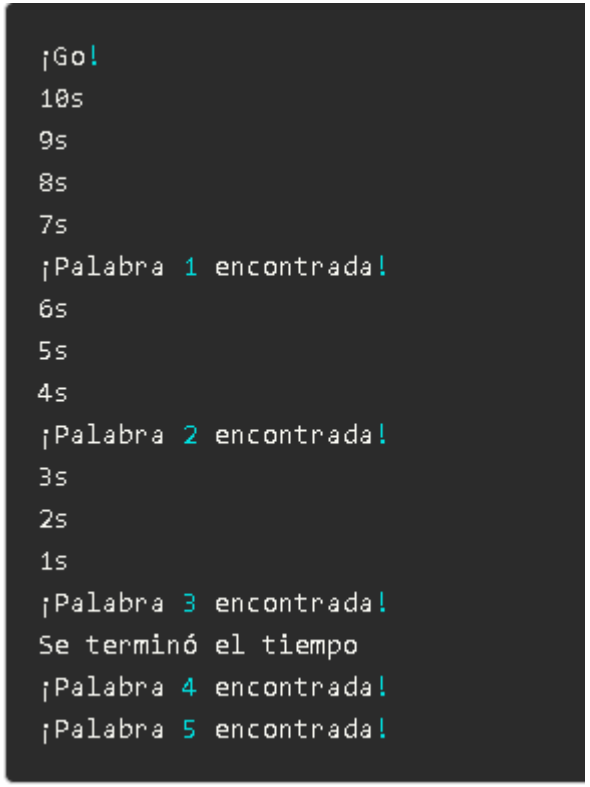
Iniciar Una Corrutina Con runBlocking()

- runBlocking() inicia una nueva corrutina y bloquea su hilo contenedor, hasta que se ejecuten todas las sentencias de su bloque de código.

- Por ejemplo, como faltaron 2 palabras en el ejemplo anterior, podemos usar runBlocking() al final para crear una corrutina que brinde unos 600 milisegundos más, mientras el usuario encuentra las palabras

```
// 4. Tiempo fuera
println("Se terminó el tiempo")
runBlocking {
    delay(600)
}
```

Al ejecutar el programa verás que se termina la corrutina inicial.



```
fun main() = runBlocking<Unit> {
    // 1. Inicio
    println("¡Go!")

    // 2. Buscar palabras en el background
    GlobalScope.launch {
        (1..5).forEach {
            delay(300)
            println("¡Palabra $it encontrada!")
        }
    }

    // 3. Iniciar temporizador en foreground
    for (i in 10 downTo 1) {
        println("${i}s")
        delay(100)
    }

    // 4. Tiempo fuera
    println("Se terminó el tiempo")

    delay(600)
}
```

También se puede completar todo en el mismo ejemplo.
De esta forma tratamos a main() como corrutina y llamamos a delay() para la suspensión en el temporizador.

Por otro lado, ya que Unit puede ser inferido por el compilador, puedes omitirlo de la declaración:

```
fun main() = runBlocking { }
```

Job

- Hasta el momento usábamos delay(600) para conseguir el resultado de la cuenta regresiva y la búsqueda de palabras.
- Esto permitió cohesionar ambos resultados sin problemas.
- Sin embargo, el método Job.join() llega al mismo resultado por nosotros.

```
fun main() = runBlocking {
    // 1. Inicio
    println("¡Go!")

    // 2. Buscar palabras en el background
    val job= GlobalScope.launch {
        (1..5).forEach {
            delay(300)
            println("¡Palabra $it encontrada!")
        }
    }

    /*...*/

    // Unir
    job.join()
}
```

El método join() suspende la corrutina principal hasta que la corrutina asociada a job se complete.

Alcance de las corrutinas

- El uso de GlobalScope.launch() crea corrutinas de nivel superior.
  - Esto quiere decir que tienen la capacidad de vivir hasta que termine la aplicación y no pueden ser canceladas prematuramente .

- Para reducir este alcance, Kotlin nos permite crear los espacios donde queremos que se dé la concurrencia.

- Por ejemplo, cuando expresamos a main() como una corrutina con runBlocking, automáticamente se creó un alcance CoroutineScope para su bloque de código.

- Por lo que hasta que las corrutinas en el interior de main() no se ejecuten, este no terminará.

- Aprovechando esto, podemos omitir la llamada de GlobalScope y de join(), ya que ahora la búsqueda de palabras hace parte del mismo alcance.

```
fun main() = runBlocking {
    // 1. Inicio
    println("¡Go!")

    // 2. Buscar palabras en el background
    launch {
        (1..5).forEach {
            delay(300)
            println("¡Palabra $it encontrada!")
        }
    }

    // 3. Iniciar temporizador en foreground
    for (i in 10 downTo 1) {
        println("${i}s")
        delay(100)
    }

    // 4. Tiempo fuera
    println("Se terminó el tiempo")
}
```

Iniciar Una Corrutina Con async()

- La función async{} crea una corrutina y retorna su resultado futuro como una instancia de Deferred<T>.

- Deferred<T> posee una función miembro denominada await(), la cual espera hasta que se ejecuten las sentencias (sin bloquear un hilo) y así entregar el valor de la corrutina.

Ejemplo

- Busquemos el tiempo total empleado por el usuario encontrando las palabras.
- Necesitaremos retornar un valor Long con los milisegundos que le tomó.

Solución

- Para solucionarlo:
  - Cambia launch{} por async{}
  - Declara y asigna el valor de async{} a una variable
  - Toma el tiempo inicial
  - Usa como valor final la resta del tiempo final del inicial
  - Obtén el tiempo con await() e imprímelo

```kotlin
import kotlinx.coroutines.*

fun main() = runBlocking {

    val totalTime = async {
        val t0 = System.currentTimeMillis()
        (1..5).forEach {
            delay(300)
            println("¡Palabra $it encontrada!")
        }
        System.currentTimeMillis() - t0
    }

    println("Tiempo empleado: ${totalTime.await()}")
}
```

```
¡Palabra 1 encontrada!
¡Palabra 2 encontrada!
¡Palabra 3 encontrada!
¡Palabra 4 encontrada!
¡Palabra 5 encontrada!
Tiempo empleado: 1512
```

La sentencia println() no se ejecuta hasta que await() tenga el resultado diferido.
Sin embargo el hilo no es bloqueado, solo se suspende la corrutina que actúa como alcance.

Iniciar Corrutina Con Alcance Personalizado

- Crea tu propio alcance para correr corrutinas, con la función de suspensión coroutineScope{}.
  - Se ejecutará hasta que todas sus sentencias sean completadas.
- A diferencia de runBlocking{}, coroutineScope{} no bloquea un hilo, si no que se suspende.

```
fun main() = runBlocking {

    coroutineScope {
        val totalTime = async {
            val t0 = System.currentTimeMillis()
            (1..5).forEach {
                delay(300)
                println("¡Palabra $it encontrada!")
            }
            System.currentTimeMillis() - t0
        }.await()

        println("Tiempo empleado: ${totalTime}")
    }
}
```

Funciones de suspensión

- Una función de suspensión o suspending function se caracteriza por ejecutarse dentro de una corrutina o al interior de otra función de suspensión.
  - El modificador suspend en la declaración de la función para marcarla como suspendible.

- Obviamente, si intentas llamarlas desde otro contexto, obtendrás un error de compilación.

Ejemplo

- Movemos las instrucciones de búsqueda del constructor sync{} a una suspending function llamada userSearchWords()

```
private suspend fun userSearchWords(): Long {
    val t0 = System.currentTimeMillis()
    (1..5).forEach {
        delay(300)
        println("¡Palabra $it encontrada!")
    }
    return System.currentTimeMillis() - t0
}
```

Al usar suspend es posible llamarla al interior del constructor de corrutina:

```
import kotlinx.coroutines.*

fun main() = runBlocking {

    val totalTime = async {
        userSearchWords()
    }.await()
    println("Tiempo empleado: ${totalTime}")
}
```

Kotlin

Flujos

En materia de corrutinas, un *flujo* es un tipo que puede emitir varios valores de manera secuencial, en lugar de *suspender funciones* que muestran solo un valor único.

Un flujo se puede usar, por ejemplo, para recibir actualizaciones en vivo de una base de datos.

Flow

- Flow es un componente de la librería de corrutinas que nos permite implementar la programación reactiva.
- Es el sustituto natural de RxJava.

- Los Flows son secuencias asíncronas.

Flow son lazy

- Esto significa que hasta que alguien no necesita los valores del Flow, las operaciones No se ejecutan (Promises vs Observables)
- Se les conoce como flujos fríos. No proveen datos hasta que alguien los pida.
- Es importante entender esto, porque **si un Flow realiza operaciones pesadas, estas se van a repetir cada vez** que alguien recolecte sus valores.

Asíncronos

- A diferencia de las secuencias, que se procesan un elemento detrás de otro, en Flow no necesariamente pasa esto.
- Puede pasar un tiempo largo entre que nos llegue un valor y el siguiente.
- Es por eso que normalmente no los ejecutaremos en el hilo principal.
- Es habitual que todo esto suceda en el contexto de corrutinas.

Secuenciales

- Esto quiere decir que si un Flow va a generar x elementos, y estos consisten en un procesamiento pesado, se van a ejecutar uno detrás de otro: hasta que no acabe el anterior no empezará el siguiente.
- Esto que muchas veces es una ventaja, en ocasiones puede ser un inconveniente.

Secuencial. Inconveniente?

- Haceos 10 peticiones a servidores y que cada una es independiente de la anterior.
- Aún así tendrías que esperar a que la anterior acabe para lanzar la siguiente.
- Esto se puede modificar.

Construir flows

- **asFlow()**
- Quizá esta es la forma más sencilla de generar un Flow.
- Todos las colecciones, incluidas las secuencias, se pueden convertir en un Flow usando esta función

```
val flow = listOf(1, 2, 3, 4).asFlow()
```

Construir flows

- **flowOf()**
- Se genera un Flow con una secuencia de valores predefinidos, el equivalente a *listOf()* o *sequenceOf()*

```
val flow = flowOf(1, 2, 3, 4)
```

Construir flows

```
flow {
    for (i in (0..3)) {
        delay(200)
        emit(i)
    }
}
```

- **flow { }**
- El más versátil de todos.
- Creamos un bloque *flow { }* y añadimos valores con la función *emit()*.
- Además, aquí sumamos la ventaja de que este bloque recibe un contexto de corrutinas, por lo que podemos llamar a funciones *suspend* sin ningún problema dentro del mismo

Operadores

- Los Flows pueden transformarse igual que las colecciones.
- Podemos filtrar, mapear, combinar, transformar… y un amplio número de operaciones que te permiten adaptar esos flujos a las necesidades que tengas en el lugar donde los utilizas.
- Al igual que con las secuencias, hay dos tipos de operadores

Operadores intermedios

- Son operadores que no lanzan ninguna operación.
- Lo que hacen es devolver un nuevo Flow que es la combinación del anterior con la nueva operación.
  - Filter
  - Map

```
makeFlow()
    .filter { it % 2 == 0 }
```

```
makeFlow()
    .filter { it % 2 == 0 }
    .map { "Value is $it" }
```

Operadores intermedios

- Pero hay un operador especialmente interesante, que es *transform()*.
- Nos permite hacer transformaciones todo lo complejas que necesitemos.
- Lo único que tenemos que hacer es llamar a *emit()* con los valores que queramos devolver:

```
makeFlow()
    .transform { value ->
        emit(value)
        emit(value * value)
    }
}
```

Operadores intermedios

- También se pueden combinar varios Flows con operaciones como *zip()* o *combine()*

```
val flow1 = flowOf(1, 2, 3, 4)
val flow2 = flowOf("1", "2", "3", "4")

flow1.zip(flow2) { a, b -> "$a -> $b" }
```

Operadores terminales

- Estos sí que lanzan la ejecución y hacen que se comience la producción de valores y estos sean emitidos.
- El operador terminal más habitual es *collect()*, que indica al Flow que ya hay alguien al otro lado (el recolector) esperando resultados, y que puede empezar a emitirlos.

```
makeFlow()
    .collect { print(it) }
```

Pero no solo hay este, tenemos varios más como *toList()*, *toSet()*, *first()*, *single()*, *reduce()* o *fold()*

Restricciones

- La primera restricción es que no podemos cambiar de contexto dentro del código de un flow.
- Si hacemos este, tendremos una excepción:

```
fun makeFlow() = flow {
    withContext(Dispatchers.IO) {
        for (i in (0..3)) {
            delay(200)
            emit(i)
        }
    }
}
```

Flow siempre va a ejecutarse en el contexto de la corrutina que lo lanzó.

Restricciones

- Para utilizar Flow en un contexto diferente de la corrutina que podemos usar la función *flowOn()*:

```
makeFlow()
    .flowOn(Dispatchers.IO)
    .collect { print(it) }
```

Excepciones

- No se debe capturar excepciones dentro de los flows, para no ocultarlas y que el resto del código no se entere.
- Hay una función especial para esto:

```
makeFlow()
    .catch { throwable -> println(throwable.message) }
    .collect { print(it) }
}
```

Si no capturamos la excepción, esta se seguirá propagando normalmente, como ocurre con el resto de componentes de corrutinas.