Programación asíncrona en Android

Kotlin Coroutines

{desafío} latam_

Utilizar elementos de la programación asíncrona acorde al lenguaje Kotlin para dar solución a un problema

- Unidad 1: Kotlin para el desarrollo de aplicaciones.
- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Reconocer los aspectos fundamentales del patrón de promesas para la ejecución asíncrona*

¿Recuerdas lo que significa programación asíncrona?

{desafío}
latam_

/* El patrón de promesas */

Future o Promise

La idea detrás de **future** o **promise** (también hay otros términos a los que se puede hacer referencia según el idioma/plataforma) es que cuando hacemos una llamada, se nos promete que en algún momento regresará con un objeto llamado Promesa, que luego se puede operar.

Promise ejemplo:

```
fun postItem(item: Item) {
    preparePostAsync()
        .thenCompose { token ->
            submitPostAsync(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }

}

fun preparePostAsync(): Promise<Token> {
    // hace un request el cual se completará después
    return promise
}
```

En este ejemplo podemos ver que la función **postItem()** ejecuta **preparePostAsync()** la cual retorna un tipo Promise. Esto permite que se pueden usar funciones definidas en Promise como `.thenCompose` y `.thenAccept`

/* ¿Qué son las coroutines en Kotlin? */

Coroutines

Una coroutine o corrutina es un patrón de diseño de simultaneidad, que se puede usar en Android, para simplificar el código que se ejecuta de forma asíncrona. Las coroutines se agregaron a Kotlin en la versión 1.3 y se basan en conceptos establecidos de otros lenguajes.

En Android, las corrutinas ayudan a administrar tareas de ejecución prolongada que, de lo contrario, podrían bloquear el hilo principal o Main Thread y hacer que la aplicación deje de responder. Actualmente, más del 50 % de los desarrolladores profesionales que usan co-rutinas informaron haber visto una mayor productividad.

Ventajas de Coroutines

Coroutines es la solución recomendada para la programación asíncrona en Android. Las características notables incluyen lo siguiente:

Ligero

Se pueden ejecutar muchas corrutinas en un solo subproceso debido a la compatibilidad con la suspensión, que no bloquea el subproceso donde se ejecuta la corrutina. La suspensión ahorra memoria sobre el bloqueo mientras admite muchas operaciones simultáneas.

Menos fugas de memoria

Utilice la simultaneidad estructurada para ejecutar operaciones dentro de un ámbito.

Soporte de cancelación incorporado

La cancelación se propaga automáticamente a través de la jerarquía de rutinas en ejecución.

Integración con Jetpack

Muchas bibliotecas de Jetpack incluyen extensiones que brindan soporte completo para corrutinas. Algunas bibliotecas también proporcionan su propio ámbito de rutina que puede usar para la simultaneidad estructurada.

Coroutines - primeros pasos

Antes de comenzar a trabajar con coroutines, lo primeros que debemos hacer es habilitarlas en nuestro proyecto, para esto, debemos agregar la siguiente línea de código a nuestro `build.gradle`

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

Coroutines - primeros pasos: Scope

Supongamos que tenemos una función en el viewmodel la cual ejecuta una coroutine, para que funcione debe cumplir con algunos requisitos

Ejemplo:

```
suspend fun someFunction(id: Int) {
        viewModelScope.launch {
            ...
        }
    }
```

Una coroutine requiere de un **scope** para ejecutarse:

- Si se ejecuta en el viewModel → **`viewModelScope`**
- Si se ejecuta en la vista (Activity/Fragment) → **`lifecycleScope`**
- Si se ejecuta fuera del viewModel o de la vista → **`withContext`**`(Dispatchers.IO){ }`

IMPORTANTE: el Dispatchers puede cambiar dependiendo del thread en el que queramos que se ejecute la coroutine

Coroutines - primeros pasos: Context y Dispatchers

- El **Context** de una coroutine es un conjunto de varios elementos. Los elementos principales son el **Job** de la **coroutine.**

- El Context de coroutine incluye un **Dispatcher** de coroutine el cual determina qué subproceso o subprocesos utiliza la coroutine correspondiente para su ejecución.

- El despachador de rutinas puede limitar la ejecución de rutinas a un subproceso específico, enviarlo a un grupo de subprocesos o dejar que se ejecute sin restricciones.

Demostración
"Hello Coroutine World"

Demostración - Coroutine 1

A continuación veremos algunos ejemplos de coroutine.
La siguiente función es una coroutine que, dos segundos después de ejecutarse, muestra el mensaje "Hello Coroutine World" en el logcat:

```
fun someCoroutine() {
    viewModelScope.launch {
        delay(2000)
        println("Hello Coroutine World")
    }
}
```

Esta función se ejecuta en el Main Thread, ya que no se especifica algo diferente, y puesto que fue creada en el viewModel, hace uso del viewModelScope.

Demostración - Coroutine 2

Supongamos que queremos ejecutar la misma función, pero en un thread diferente, esto podemos hacerlo si especificamos el Dispatcher.

`Dispatchers.IO`: se usa cuando se necesita ejecutar tareas que pueden tomar tiempo en completarse, como consultas a bases de datos http request.

```
fun someCoroutine() {
    viewModelScope.launch(Dispatchers.IO) {
        delay(2000)
        println("Hello Coroutine World")
    }
}
```

Demostración - Coroutine 3

¿Qué pasa si creamos esta función en un fragment por ejemplo, que scope debemos usar?

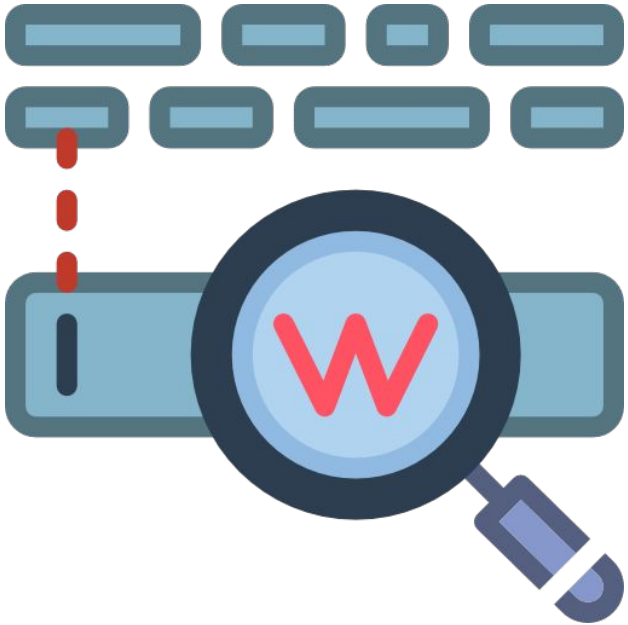
```
fun someCoroutine() {
    lifecycleScope.launch {
        delay(2000)
        println("Hello Coroutine World")
    }
}
```

Para este caso debemos usar `lifecycleScope`, adicionalmente, `lifecycleScope` ofrece otras opciones para determinar en qué momento del ciclo de vida se ejecuta la coroutine:

- `launchWhenCreated`
- `launchWhenStarted`
- `launchWhenResumed`

/* Kotlin Coroutines, palabras reservadas*/

Palabras reservadas



- **suspend:** es un tipo de función que puede ser iniciada, pausada y resumida, una suspend function solo puede ser llamada desde una coroutine o desde otra suspend function.

- **Scope**: se puede definir como las restricciones en la cual se ejecuta la coroutine, los scope pueden ser: **GlobalScope**, **LifeCycleScope** y **ViewModelScope.**

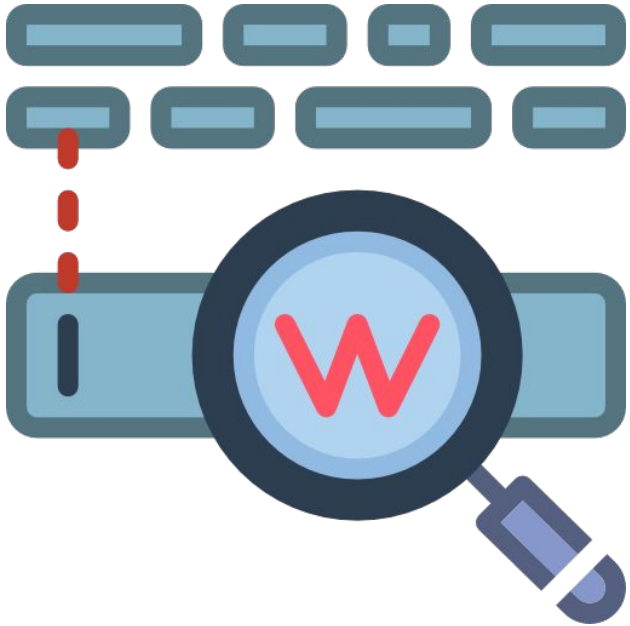
- **Dispatcher**: indica el thread en el que se ejecuta la coroutine. Los dispatchers pueden ser: **Main** Dispatcher, **IO** Dispatcher, **Default** Dispatcher, **Unconfined** Dispatcher.

Palabras reservadas

- **Context**: requerido para ejecutar una coroutine y es representado por un valor tipo CoroutineContext.

- **Channels**: proporcionan una forma de transferir un flujo de valores.

- **Job**: es algo que se puede cancelar con un ciclo de vida que culmina con su finalización. Ejecuta un bloque de código específico y se completa al finalizar este bloque.



Async y Await

Kotlin, como lenguaje, proporciona sólo API mínimas de bajo nivel en su librería estándar para permitir que otras librerías utilicen coroutines. A diferencia de muchos otros lenguajes con capacidades similares, **async** y **await** no son palabras clave en Kotlin y ni siquiera forman parte de su biblioteca estándar.

El concepto de función de suspensión o “suspend function” de Kotlin proporciona una abstracción más segura y menos propensa a errores para las operaciones asincrónicas que los futuros y las promesas.

/* Concurrencia en coroutines */

¿Qué es Concurrencia?

Que es **concurrencia**: Un lenguaje de programación concurrente se define como uno que utiliza el concepto de ejecución simultánea de procesos o hilos de ejecución como medio para estructurar un programa. Un lenguaje paralelo es capaz de expresar programas que son ejecutables en más de un procesador.

Un ejemplo sencillo para visualizar lo que quiere explicar la concurrencia es el siguiente:

```
fun main() {
    test1WithThread()
    test2WithThread()
}

fun test1WithThread() {
    println("Start thread test 1:
${threadName()}")
    Thread.sleep(500)
    println("End thread test 1: ${threadName()}")
}

fun test2WithThread() {
    println("Start thread test 2:
${threadName()}")
    Thread.sleep(1000)
    println("End thread test 2: ${threadName()}")
}
```

¿Podrías explicar con tus palabras qué es el patrón de promesas?

Próxima sesión...

- *Utilizar Coroutines para el manejo asíncrono acorde al lenguaje Kotlin.*

{desafío}
latam_

Academia de
talentos digitales