Corrutinas

¿Qué son?

Introducción

Una corrutina es un patrón de diseño de concurrencia que permite simplificar la ejecución de tareas asincrónicas (y también sincrónicas). Las corrutinas sólamente se pueden utilizar con el lenguaje Kotlin a partir de la versión 1.3.
Esta solución es la recomendada por Android para la programación asincrónica debido a las ventajas que brinda:

- Son livianas: gracias al concepto de *suspensión* se pueden correr muchas corrutinas en un mismo hilo de ejecución.
- Baja probabilidad de memory leaks: corren dentro de un *scope* (ámbito) definido.
- Soporte de Cancelación: existen distintas estrategias que permiten controlar la propagación de cancelaciones de las corrutinas de manera jerárquica.
- Integración con Jetpack: muchas librerías de Jetpack incluyen extensiones que proveen una incorporación completa para su utilización.

Integración

Para poder utilizar las corrutinas en el proyecto de Android se deberá agregar la dependencia en el archivo build.gradle:
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.4.1'
Extra: para asociar la corrutina a un ciclo de vida
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.3.1'

Conceptos claves

Suspend

Es una palabra reservada que se usa como prefijo para las funciones. La palabra *suspend* indica que la función puede ser pausada y resumida en un punto de tiempo. Permite ejecutar operaciones y esperar a que sea completada sin bloquear el hilo principal. Ejemplo:

```
private suspend fun hacerAlgo() {

}
```

Este tipo de funciones pueden ser llamadas dentro de una corrutina o también desde otra *suspend function*.

Job

El Job es un objeto al que se le asocian las corrutinas para poder manipularlas de manera conjunta. El mismo posee un ciclo de vida que finaliza cuando se ejecutan las corrutinas que

posee, cuando se cancela, o también cuando falla. En el patrón de Rx se lo podría comparar con el *CompositeDisposable*.
El Job es creado al momento de crear la corrutina mediante los métodos *launch* o *async*. *Más adelante se verán estos conceptos*.
Si dentro del Job ocurre una excepción distinta a *CancellationException* se cancelarán la ejecución de las corrutinas. Para evitar esto se utiliza el *SupervisorJob*, en donde se le deberá pasar por parámetro los Jobs que contendrá y que correrá. Esto permitirá que cada Job pueda fallar independientemente de otro.
Para cancelar un Job se utiliza el método *cancel*.

Dispatchers

Los *dispatchers* especifican en qué hilo la operación va a ocurrir. Estos pueden ser definidos al momento de lanzar la corrutina mediante el *builder* de corrutinas, o también llamando a la función *withContext*. En Rx se lo podría comparar con los *Schedulers*.
Entre ellos se encuentran:

- Main: hilo de la vista.
- Default: dispatcher predeterminado que se utiliza si no se especifica otro. Está optimizado para el trabajo intenso de la CPU.
- IO: usualmente se utiliza para realizar llamadas a servicios web, interacciones con base de datos, etc.

Scope

El *Scope* es el ámbito donde las corrutinas tendrán efecto. Se pueden crear propios *Scopes*, o utilizar los ya creados. Dentro de ellos se pueden encontrar:

- GlobalScope: es un ámbito general que posee un *builder* para crear nuevas corrutinas. Mayormente en Android no se sugiere utilizar el GlobalScope, ya que la corrutina está atada a finalizar su ejecución o a que la aplicación se cierre. Por lo tanto si se quiere correr una dentro de una Activity, y la misma es destruida, la corrutina continuará ejecutándose.
- LifecycleScope: es el ámbito que se ata al ciclo de vida. Si se corre una corrutina en este ámbito dentro de una Activity, al destruirse la misma, la corrutina también lo hará.
    - Fragmento: se podrá acceder a este ámbito mediante *viewLifecycleOwner.lifecycleScope*
    - Activity: se podrá acceder a este ámbito llamando directamente a *lifecycleScope*
- ViewModelScope: es un ámbito que está atada a la vida del ViewModel. En el curso de Jetpack se verá acerca de este concepto.

WithContext

La función *withContext* permite ejecutar un bloque de código dentro de una función de suspensión en un Dispatcher definido. Este Dispatcher se define como primer parámetro de dicha función, y como segundo parámetro se definirá el bloque de código a ejecutar.

Mayoritariamente esta función se utiliza para moverse entre distintos hilos, o también para definir una función que corra en un hilo en particular.
Ejemplo 1: definición de función de suspensión para que corra en el hilo IO

```
private suspend fun hacerAlgo() = withContext(Dispatchers.IO) {
    // Ejecutar funcionalidad dentro del hilo IO
}
```

Ejemplo 2:

```
lifecycleScope.launch(Dispatchers.Main) {  this: CoroutineScope
    // Corre en el hilo principal
    withContext(Dispatchers.IO) {  this: CoroutineScope
        // Corre en el hilo secundario
    }
}
```

Coroutine Builders

Introducción

Los constructores de corrutinas permiten crear e iniciar corrutinas a partir de funciones de extensión. Dentro de ellos se encuentran *launch*, *async* y *runBlocking*.

Launch

La manera más simple de crear una corrutina es llamando al método *launch* dentro de un ámbito específico. Lanza una nueva corrutina que no bloquea el hilo actual y devuelve su referencia como un *Job*. Esta corrutina es cancelada cuando el resultado del Job es cancelado. Por lo general este tipo de builder se utiliza para ejecutar código sin estar pendiente del resultado.
Ejemplo 1:

```
lifecycleScope.launch(Dispatchers.IO) {
    actualizarLibro(libro)
}
```

Ejemplo 2:

```
lifecycleScope.launch(Dispatchers.IO) {
    delay( timeMillis: 1500)
    println("Hola")
}
println("Educación IT")
```

El resultado de este ejemplo será:
I/System.out: Educación IT
I/System.out: Hola

Async

El builder *async* crea una nueva corrutina y devuelve el resultado con una función de suspensión llamada *await*. Este se utiliza para correr tareas en paralelo en donde la finalización de una sea dependiente de la otra, ya que esta función permite esperar mediante el método *await* la ejecución del código siguiente. La misma deberá ser ejecutada dentro de una corrutina o una función de suspensión.

Ejemplo: en este caso se crea una corrutina mediante el método launch y luego se aplica el *async* junto con el *await*.

```
lifecycleScope.launch(Dispatchers.IO) {
    val libro = async { obtenerLibro() }
    val libroObtenido = libro.await()
    libroObtenido.autor = "Nicolas"
    actualizarLibro(libroObtenido)
}
```

RunBlocking

Bloquea el hilo de donde es invocado hasta que la corrutina es completada. Es decir que se ejecutará en un orden lineal.

Ejemplo 1:

```
runBlocking(Dispatchers.IO) {
    delay( timeMillis: 2000)
    actualizarLibro(libro)
}
```

Ejemplo 2:

```
runBlocking(Dispatchers.IO) {
    delay( timeMillis: 1500)
    println("Hola")
}
println("Educación IT")
```

El resultado de este ejemplo será:

I/System.out: Hola

I/System.out: Educación IT

Manejo de errores

Introducción

Para poder capturar los errores y excepciones que puedan llegar a ocurrir en la ejecución de la corrutina se utilizan los bloques de try/ctach.

Otra manera en la cual se pueden manejar los errores es a través de la utilización del CoroutineExceptionHandler.

Launch

La excepción será arrojada de manera inmediata, por lo que se deberá agregar el bloque de try/catch al momento de haber iniciado la corrutina.

```
lifecycleScope.launch(Dispatchers.IO) {
    try {
        actualizarLibro(libro)
    } catch (e: Exception) {
        // Manejar excepción
    }
}
```

Async

En este builder la excepción va a ser retenida hasta que el método *await* es invocado. Por lo tanto se deberá agregar el bloque try/catch en el momento en el que se llama al método *await*.

```
lifecycleScope.launch(Dispatchers.IO) {
    val libro = async { obtenerLibro() }

    try {
        val libroObtenido = libro.await()
        libroObtenido.autor = "Nicolas"
        actualizarLibro(libroObtenido)
    } catch (e: Exception) {
        // Manejar excepción
    }
}
```