# Código asíncrono con corrutinas de Kotlin en Android

**Gunar Percy Ortiz Chura**
Universidad Mayor de San Andrés
Facultad de Ciencias Puras y Naturales
Carrera de Informática
La Paz, Murillo
Bolivia
Email: gunarortiz@gmail.com

## RESUMEN
Es muy común tener que escribir código asíncrono en el desarrollo de aplicaciones móviles. Los métodos clásicos con los cuales se suele manejar esto produce que el código sea difícil de entender y probar. En la versión 1.1 de Kotlin se agregaron las corrutinas, cuya función es facilitarnos el manejo del código asíncrono, haciendo que este tome una forma secuencial facilitando la legibilidad del código, además, optimiza el rendimiento de los "hilos" que se crean a partir de estas tareas. El documento aborda la forma en la que se debe trabajar con las corrutinas.

## Palabras clave
Corrutinas, Asíncrono, Kotlin, Apps, Android.

## 1 INTRODUCCIÓN
Al momento de desarrollar aplicaciones Android con Kotlin estamos acostumbrados a programar de manera secuencial, ya que que la arquitectura de dichas aplicaciones así lo requieren cuando de cosas básicas se trata (realizar cálculos simples, acceder a los sensores, dibujar la UI, etc). Avanzando un poco en complejidad nos topamos con que muchos de los datos que necesita nuestra aplicación para funcionar no están disponibles inmediatamente, ya sea porque se necesita realizar un cálculo previo, o estos se encuentran accesibles a través de la red.

Aquí es donde un programador acostumbrado a llevar el desarrollo de una app de manera secuencial debe adaptarse y armarse de herramientas para trabajar de esta forma. La opción más usada son los callbacks, que no son más que funciones que son llamadas cuando los datos están listos, existen muchos problemas asociados a esta técnica como ser: dificultad en la legibilidad del código, dificultad en debuggear la aplicación, el mal manejo de estos puede provocar el callback hell[1].

Otra técnica usada son los Threads, los cuales exigen un conocimiento más profundo de cómo se llevan a cabo los procesos en el SO de Android, la principal desventaja de estos es que generalmente un Threads suele ser mucho para una tarea sencilla o muy poco para una tarea compleja, haciendo que el diseño óptimo de estos sea una etapa muy importante, morosa y complicada. El equipo de Kotlin implementó las corrutinas pensando en todos los problemas que significaba tener que trabajar con procesos asíncronos.

Las corrutinas son muy fáciles de usar gracias a su sintaxis, lo que facilita la lectura y mantenimiento del código, estas vuelven el código asíncrono en un código secuencial pausando la ejecución del programa en lugares específicos hasta obtener los resultados, además están optimizados lo suficiente como para no considerar el uso de los Threads. Este documento aborda la forma en la que se trabaja con las corrutinas y las razones por las cuales son la mejor opción para lidiar con el código asíncrono.

## 2 ¿QUÉ ES UNA CORRUTINA?
Podemos encontrar dichas corrutinas en kotlinx.coroutines que es una biblioteca desarrollada por JetBrains. Contiene una serie de corrutinas primitivas ya habilitadas y de alto nivel.[1]

En esencia una corrutina en un hilo liviano [2] que en lugar de bloquear el hilo principal, un proceso o la ejecución de un programa, permite que este se siga ejecutando; también proporciona una abstracción más segura y menos propensa a errores para operaciones asíncronas.

Las alternativas a las corrutinas llegan a ser: Threads, Callbacks, RX (reactive, extensions), futures y promises.

## 3 CREACIÓN DE UNA RUTINA
Para utilizar las corrutinas se debe agregar una dependencia en el `kotlinx-coroutines-core`.

### Maven
Agregue dependencias (también puede agregar otros módulos que necesite):

```
<dependency>
<groupId>org.jetbrains.kotlinx</groupId>
<artifactId>
kotlinx-coroutines-core
</artifactId>
<version>1.3.0-RC2</version>
</dependency>
```

Y asegúrese de utilizar la última versión de Kotlin:

```
<properties>
```

[1] El Callback Hell se produce cuando encadenamos muchas operaciones asíncronas seguidas.

```
<kotlin.version>1.3.41</kotlin.version>
</properties>
```

### Gradle

Agregue dependencias (también puede agregar otros módulos que necesite):

```
dependencies {
implementation
 'org.jetbrains.kotlinx:
 kotlinx-coroutines-core:1.3.0-RC2'
}
```

Y asegúrese de utilizar la última versión de Kotlin:

```
buildscript {
ext.kotlin_version = '1.3.41'
}
```

Asegúrese de tener un `jcenter()` o `mavenCentral()` en la lista de repositorios[1]:

```
repository {
jcenter()
}
```

### Gradle Kotlin DSL

Agregue dependencias (también puede agregar otros módulos que necesite):

```
dependencies {
implementation("org.jetbrains.kotlinx:
kotlinx-coroutines-core:1.3.0-RC2")
}
```

Y asegúrese de utilizar la última versión de Kotlin:

```
plugins {
kotlin("jvm") version "1.3.41"
}
```

Las corrutinas se lanzan con el coroutine builder de lanzamiento en un contexto de algunos CoroutineScope.

### Ejemplo

```
import kotlinx.coroutines.*

fun main() {
GlobalScope.launch {
delay(1000L)
println("World!")
}
println("Hello,")
Thread.sleep(2000L)
}
```

Verá el siguiente resultado:

```
Hello,
World!
```

Aquí estamos lanzando una nueva rutina en el GlobalScope , lo que significa que la vida útil de la nueva rutina está limitada sólo por la vida útil de toda la aplicación. [3]

## 4 CANCELACIÓN DE UNA RUTINA

Existe la posibilidad de que cuando una corrutina de larga ejecución se esté llevando a cabo el usuario deje la pantalla o que ya no requiera los resultados de este proceso, para cancelar un corrutina disponemos del método `cancel()` [4]. Este método cancelará la corrutina sin importar el estado en el que se encuentre. Para cancelar el proceso y esperar su finalización tenemos el método `cancelAndJoin()`.

Es importante conocer el ciclo de vida de los activities en Android, ya que el SO suele impedir que muchas tareas se ejecuten de manera continua e ilimitada, a menos que se lo hagamos saber de qué manera debe ejecutar estos procesos y bajo qué circunstancias, pasan a ser totalmente administrados por el SO.

## 5 TIEMPO DE ESPERA

Podemos definir un tiempo de espera para controlar de alguna manera a las corrutinas, si esta excede el tiempo que nosotros definimos lanzará la excepcion `TimeoutCancellationException` [4].

```
withTimeout(1300L) {
repeat(1000) { i ->
println("I'm sleeping $i ...")
delay(500L)
}
}
```

el metodo `withTimeout` recibe como parámetro el tiempo máximo de espera para que la rutina termine.

## 6 COMPONER FUNCIONES DE SUSPENSIÓN

La premisa del documento es que las corrutinas nos ayudan a tratar el código asíncrono de manera secuencial, pero ¿cómo logramos esto?.

Primero debemos definir una función de tipo suspend [5], para el ejemplo tendremos dos funciones y haremos que cada una demore un segundo.

```
suspend fun doSomethingUsefulOne(): Int {
delay(1000L)
return 13
}

suspend fun doSomethingUsefulTwo(): Int {
delay(1000L)
return 29
}
```

En el espacio de código que invoquemos esas funciones, estas deben suspender la ejecución del programa hasta que obtengan el resultado, y eso es precisamente lo que hace la función de tipo suspend

```
val uno = doSomethingUsefulOne()
val dos = doSomethingUsefulTwo()
println("The answer is ${uno + dos}")
```

cuando el programa está en la línea donde se define la variable "uno" esté de suspende hasta que la función `doSomethingUsefulOne()` termine, sucede igual con la

variable “dos” haciendo que nuestro programa se ejecute de manera secuencial sin ningún problema.

# 7 USO SIMULTÁNEO UTILIZANDO ASÍNCRONOS

Existen casos en los que por temas de optimización no es necesario que una función tenga que esperar a que otra finalice, así que podemos lanzar estas funciones en simultáneo. Acá es donde introducimos el método await, que como su traducción indica simplemente espera a que los datos esten listos, y la clase async[5] que indica que la función a la que esté referenciando se va a tardar.

```
val uno = async { doSomethingUsefulOne() }
val dos = async { doSomethingUsefulTwo() }
println("The answer is ${uno.await()
+ dos.await()}")
```

cuando se requieren los datos para imprimirlos en pantalla el método `await` hace que de ambas rutinas se inicien simultáneamente.

## 7.1 INVOCAR UNA FUNCIÓN ASÍNCRONA

También tenemos la posibilidad de iniciar una corrutina antes de necesitar los datos, para eso hacemos uso del método start. Ademas tenemos que pasar al constructor de `async`[5] un parámetro con el valor de `start = CoroutineStart.LAZY`

```
val uno=async(start=CoroutineStart.LAZY)
{ doSomethingUsefulOne() }
val dos=async(start=CoroutineStart.LAZY)
{ doSomethingUsefulTwo() }
uno.start()
dos.start()
println("The answer is ${uno.await()
+ dos.await()}")
```

aquí uno y dos se ejecutan cuando hacemos uso del método `start`, y cuando queremos imprimirlos en pantalla se espera a que estos finalicen individualmente.

# 8 CONCLUSIONES

Las corrutinas aún son bastante nuevas, y como le sucede a toda tecnología nos tomará tiempo aprender a sacarle todo el provecho.

Tal vez sea demasiado pronto para decir que las corrutinas lleguen a desfasar completamente a las herramientas que tienen el mismo propósito, pero por las características ya mencionadas sin lugar a dudas ocupara un lugar importante en el desarrollo de una aplicación móvil con Koltin en el futuro.

El código usado en este documento se encuentra en el repositorio de Kotlin [6].

# 9 AGRADECIMIENTOS

Agradezco profundamente a la organización del evento “TechZone 2019” por darme la posibilidad y las bases para realizar este documento. Además debo agradecer a la comunidad de Kotlin por haber realizado varios de los aportes que intente rescatar en este documento.

# References

[1] kotlinx.coroutines - https://github.com/kotlin/kotlinx.coroutines/blob/master/README.mdusing-in-your-projects.

[2] Coroutines Guide - https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html

[3] Coroutine Basics - https://kotlinlang.org/docs/reference/coroutines/basics.html

[4] Cancellation and Timeouts - https://kotlinlang.org/docs/reference/coroutines/cancellation-and-timeouts.html

[5] Composing Suspending Functions - https://kotlinlang.org/docs/reference/coroutines/composing-suspending-functions.html

[6] Repository Kotlin - https://https://github.com/Kotlin/kotlinx.coroutines