



Programación multimedia y dispositivos móviles

UT-3.3 Tareas en 2º plano. Corrutinas (hilos)

<https://developer.android.com/codelabs/basic-android-kotlin-compose-coroutines-kotlin-playground?hl=es-419#4>

<https://developer.android.com/kotlin/coroutines?hl=es-419>



Tareas en segundo plano en Android: Corrutinas

Todos los componentes de una aplicación Android, tanto las actividades, los servicios [sí, también los servicios], o los broadcast receivers se ejecutan en el mismo hilo de ejecución, el llamado hilo principal, main thread o GUI thread, que como éste último nombre indica también es el hilo donde se ejecutan todas las operaciones que gestionan la interfaz de usuario de la aplicación.

Es por ello, que cualquier operación larga o costosa que realicemos en este hilo va a bloquear la ejecución del resto de componentes de la aplicación y por supuesto también la interfaz, produciendo al usuario un efecto evidente de lentitud, bloqueo, o mal funcionamiento en general, algo que deberíamos evitar a toda costa.

Incluso puede ser peor, según lo que estés haciendo te puede aparecer el mensaje de «Application Not Responding» y el usuario debe decidir entre forzar el cierre de la aplicación o esperar a que termine.



Tareas en segundo plano en Android: Corrutinas

Para corregir esto debemos ejecutar en segundo plano procesos u operaciones de *“larga duración”*. ¿Qué es un proceso de larga duración? Pues cualquiera que no sepas lo que vas a tardar o cuya respuesta no depende del usuario: accesos a datos, a servicios API externos, etc, etc

Para ejecutar procesos en segundo plano o fuera del hilo principal necesitamos recurrir a nuestros viejos amigos: los **hilos**.

El trabajo con hilos en Kotlin lo vamos a hacer sin trabajar directamente con ellos, ya que Google ha desarrollado (y por lo que dicen de forma muy potente) un patrón de diseño llamado [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”



Tareas en segundo plano en Android: Corrutinas

Las corrutinas no son exclusivas de Android. Según [Wikipedia](#):

Las corrutinas son componentes [de programas informáticos](#) que permiten suspender y reanudar la ejecución, generalizando [subrutinas](#) para [la multitarea cooperativa](#) . Las corrutinas son adecuadas para implementar componentes de programas familiares, como [tareas cooperativas](#) , [excepciones](#) , [bucles de eventos](#) , [iteradores](#) , [listas infinitas](#) y [tuberías](#) .

Se han descrito como "funciones cuya ejecución se puede pausar". ^[1]

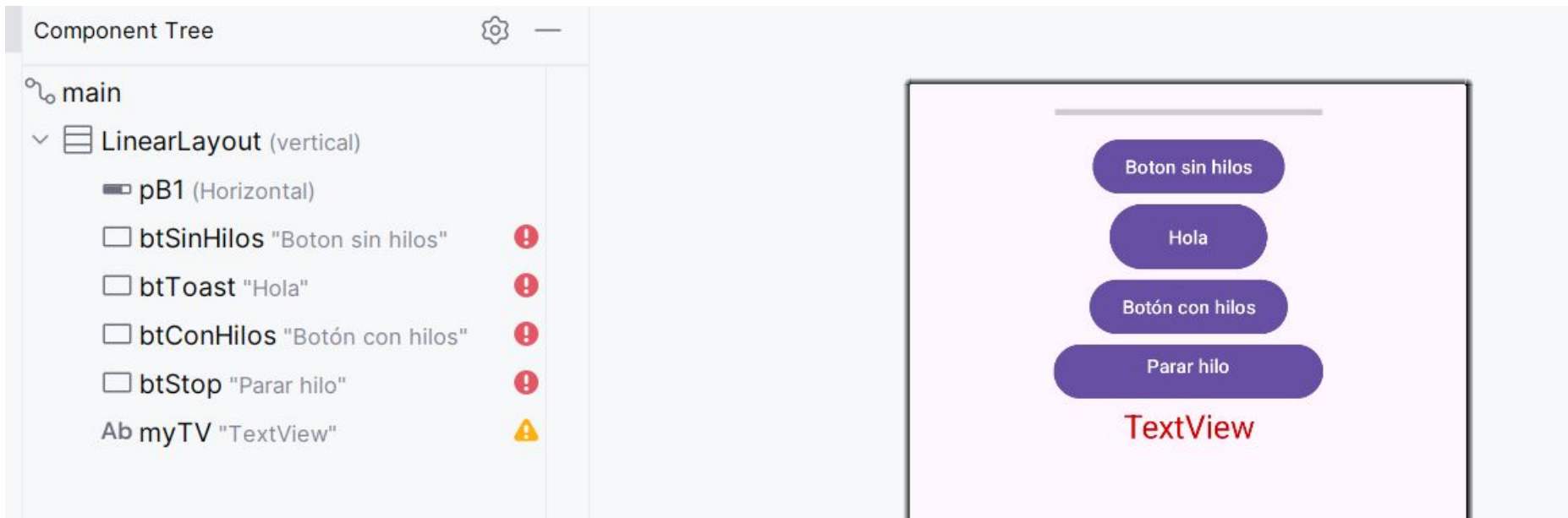


Tareas en segundo plano en Android: Corrutinas

Como siempre vamos a trabajar el concepto con un proyecto. Vamos con un proyecto nuevo. Os paso el diseño de la actividad principal donde he incluido varios botones, un textview y un control nuevo muy fácil de utilizar: la barra de progreso.

Aunque no es obligatorio hacerlo así, yo voy a trabajar en este proyecto con vinculación de vistas (binding), por lo que incluyo la sección de buildFeatures en el gradle, sincronizo y hago los cambios en la clase de kotlin para poder trabajar con el binding.

Tareas en segundo plano en Android: Corrutinas





Tareas en segundo plano en Android: Corrutinas

Para simular una operación de larga duración vamos a ayudarnos de un método auxiliar que lo único que haga sea esperar 1 segundo, mediante una llamada a `Thread.sleep()`.

```
fun tareaLarga() {  
    try {  
        Thread.sleep( millis: 1000)  
    } catch (e: InterruptedException) {  
    }  
}
```

Ahora hacemos que el primer botón ejecute este método 10 veces, de forma que nos quedará una ejecución de unos 10 segundos en total.



Tareas en segundo plano en Android: Corrutinas

En cada una de las vueltas vamos a actualizar la barra de progreso. En este método fijamos el valor inicial y final de la barra de progreso y un mensaje cuando hayamos acabado. Este método lo llamamos desde el botón sin hilos.

```
binding.btSinHilos.setOnClickListener(View.OnClickListener {  
    binding.pB1.setMax(100)  
    binding.pB1.setProgress(0)  
    for (i in 1 .. 10) {  
        tareaLarga()  
        binding.pB1.incrementProgressBy( diff: 10)  
    }  
    Toast.makeText( context: this, text: "Tarea finalizada!", Toast.LENGTH_SHORT).show()  
    binding.myTV.text = "Tarea Finalizada"  
})
```




Tareas en segundo plano en Android: Corrutinas

Ahora, antes de empezar a probar lo que está pasando vamos a añadir un método cuya única misión es generar un mensaje toast. Este método lo vamos a llamar desde el botón de “Hola”.

```
binding.btToast.setOnClickListener(View.OnClickListener {  
    Toast.makeText(context: this, text: "Hola clase de DA2D1E", Toast.LENGTH_LONG).show()  
})
```

Vamos a comprobar que está pasando. Arrancamos y le damos al “botón sin hilos”

¿Se actualiza la barra de progreso?

¿Que pasa si le damos al botón de “hola” mientras está haciendo el bucle anterior?



Tareas en segundo plano en Android: Corrutinas




Lo que ha ocurrido es que desde el momento que hemos pulsado el botón para ejecutar la tarea, hemos bloqueado completamente el resto de la aplicación, incluida la actualización de la interfaz de usuario, que ha debido esperar a que ésta termine mostrando directamente la barra de progreso completamente llena. En definitiva, no hemos sido capaces de ver el progreso de la tarea.

Pero como todo puede empeorar, al intentar pulsar el botón de “Hola” (o cualquier otra acción en realidad) ,Android nos ha advertido con un mensaje de error que la aplicación no responde debido a que se está ejecutando una operación de larga duración en el hilo principal. El usuario debe elegir entre esperar a que termine de ejecutarla o forzar el cierre de la aplicación.



Corrutinas

Las corrutinas son nuestra solución recomendada para la programación asíncrona en Android. Las funciones más importantes son las siguientes:

- **Ligereza:** Puedes ejecutar muchas corrutinas en un solo subproceso debido a la compatibilidad con la [suspensión](#) , que no bloquea el subproceso en el que se ejecuta la corrutina. La suspensión ahorra más memoria que el bloqueo y admite muchas operaciones simultáneas.
- **Menos fugas de memoria:** Usa [simultaneidad estructurada](#)  para ejecutar operaciones dentro de un permiso.
- **Compatibilidad integrada con la cancelación:** [Cancelación](#)  se propaga automáticamente a través de la jerarquía de corrutinas en ejecución.



Corrutinas: funciones de suspensión

Una de las ventajas de las corrutinas te permiten escribir tu código asíncrono de forma secuencial por lo que simplifica el código desarrollado.

Las corrutinas se basan en la idea de las **funciones de suspensión**. Estas son funciones que pueden detener la ejecución de una corrutina en cualquier punto y luego devolverle el control una vez que el resultado esté listo y la función haya terminado de hacer su trabajo.

Por lo tanto, las corrutinas son básicamente un lugar seguro donde las funciones de suspensión (normalmente) no bloquean el hilo principal.

Para parar o detener un proceso, provocar un retraso simulando una llamada a un API que tarda en responder, vamos a utilizar la función **delay()**. Es muy similar al sleep de los hilos.



Tareas en segundo plano en Android: Corrutinas

```
fun tareaLargaKotlin() {  
    delay( timeMillis: 1000)  
}
```

En realidad, `delay()` es una función de suspensión especial que proporciona la biblioteca de corrutinas de Kotlin. En este punto, se suspenderá (o detendrá) la ejecución.

Si intentas ejecutar tu programa en este momento, verás un error de compilación: Suspend function 'delay' should be called only from a **coroutine** or another **suspend function**.



```
✓ -> suspend fun tareaLargaKotlin() {  
    delay( timeMillis: 1000)  
}
```



Corrutinas: ámbito o alcance

Las corrutinas se generan dentro de un ámbito o alcance. De forma general este “alcance”, fuera de Android Studio, suele ser CoroutineScope.

Un **CoroutineScope** está vinculado a un ciclo de vida, que establece límites sobre la duración en la que estarán activas las corrutinas dentro de ese alcance. Si se cancela un alcance, el trabajo se cancela, y la cancelación se propaga a los trabajos secundarios. Si un trabajo secundario en el alcance falla con una excepción, otros trabajos secundarios y el trabajo superior se cancelan, y la excepción se vuelve a arrojar al llamador.



Corrutinas: ámbito o alcance para Android Studio

Android proporciona compatibilidad con el alcance de corrutinas en entidades que tienen un ciclo de vida bien definido, como **Activity (lifecycleScope)** y **ViewModel (viewModelScope)**. Las corrutinas que se inician dentro de estos alcances cumplirán con el ciclo de vida de la entidad correspondiente, como Activity o ViewModel.

Por ejemplo, supongamos que inicias una corrutina en un **Activity** con el alcance de corrutinas proporcionado que se denomina **lifecycleScope**. Si se destruye la actividad, se cancelará lifecycleScope, y también se cancelarán automáticamente todas sus corrutinas secundarias.

Cuando ejecutas la tarea de forma asíncrona o simultánea, hay preguntas que debes hacerte sobre cómo durante cuánto tiempo debe existir la corrutina o qué sucede si se cancela o falla con un error.



Corrutinas: ámbito o alcance para Android Studio

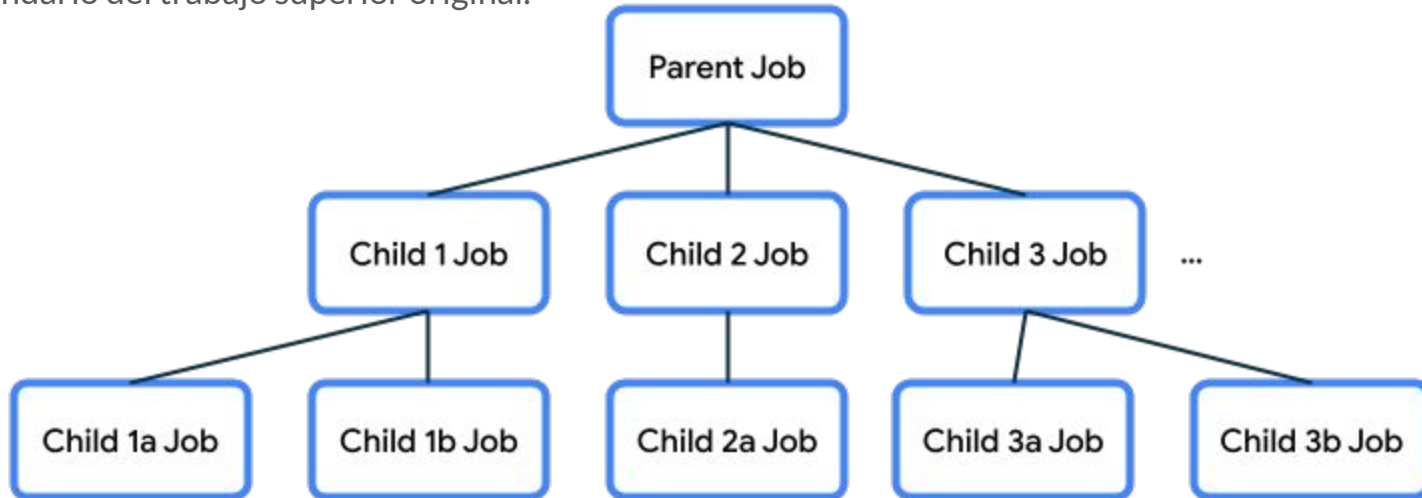
Las corrutinas se inician por tanto dentro de un ámbito o alcance. Se inician con las funciones: **launch()** y **async()**.

El método **launch** se utiliza para crear una corrutina que **no** devuelve un resultado, mientras que el método **async** se utiliza para crear una corrutina que devuelve un resultado. Ambas funciones toman un bloque de código lambda como argumento que contiene la lógica de la corrutina.

Cuando inicias una corrutina con la función, se devuelve una instancia de **Job**. Dicha instancia contiene un controlador o referencia de la corrutina para que puedas administrar su ciclo de vida

Corrutinas: jerarquía de trabajos

Desde una corrutina puedes iniciar otra: el trabajo que muestra la corrutina nueva se considera secundario del trabajo superior original.





Corrutinas: paso 1, dependencias

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

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

No olvides que siempre después de añadir una dependencia, necesitamos sincronizar.



Corrutinas: paso 2, definimos el job

Dentro de nuestra clase kotlin vamos a crearnos una variable con el job de la corrutina para una vez lanzada, poder pararla:

```
class MainActivity : AppCompatActivity() {  
    private lateinit var binding: ActivityMainBinding  
    lateinit var myCorrutina: Job
```

Corrutinas: paso 3, lanzamos la corrutina

Dentro del click del botón con hilos, vamos a ejecutar el mismo código que hemos lanzado para el botón sin hilos, pero lo hacemos dentro de una corrutina ejecutada en el ámbito de la actividad:



```
binding.btConHilos.setOnClickListener(View.OnClickListener {  
    binding.pB1.setMax(100)  
    binding.pB1.setProgress(0)  
    myCorrutina = lifecycleScope.launch {  
        for (i in 1 ≤ .. ≤ 10) {  
            tareaLargaKotlin()  
            binding.pB1.incrementProgressBy( diff: 10)  
        }  
        Toast.makeText( context: this@MainActivity, text: "Tarea finalizada!", Toast.LENGTH_SHORT).show()  
        binding.myTV.text = "Tarea Finalizada"  
    }  
    binding.myTV.text = "Tarea en curso"  
})
```



Tareas en segundo plano en Android: Corrutinas

Ahora podemos probar nuestra app y vemos que la aplicación NO se bloquea. la corrutina genera un hilo, en el que hace el bucle y actualiza la barra de progreso.

Mientras el hilo principal se ha quedado liberado y podemos pulsar el botón de Toast y nos hará caso sin “crashear” la aplicación.



Corrutinas: paso 4, paramos la corrutina

Dentro del click del botón de parar hilo, vamos a parar el trabajo que apunta a la corrutina

```
binding.btStop.setOnClickListener(View.OnClickListener {  
    binding.myTV.text = "Tarea cancelada"  
    myCorrutina.cancel()  
})
```