# Hilos

Si una aplicación realiza una operación de larga duración (como el procesamiento de datos, un bucle, acceso a la red, a base de datos, a ficheros, etc.) en el hilo principal, provocará que la interfaz no responda y que la aplicación quede bloqueada. Si la congelación dura más de cuatro segundos, el propio sistema nos mostrará el diálogo ANR (Application not responding) y nos invitará a cerrarla. Tenemos que evitar que esto suceda, ya que dará una mala imagen al usuario y es muy probable que si esto ocurre en varias ocasiones termine desinstalándola.

El problema anterior ocurre porque las aplicaciones de Android se ejecutan en un único hilo, que se usa para gestionar la interfaz y, por lo tanto, si realizamos una operación que tarde un tiempo en este mismo hilo (aunque sean segundos), estaremos bloqueando la interfaz. Para evitar esto, todas las operaciones de larga duración han de realizarse de forma asíncrona en un **hilo separado**, fuera del hilo principal de la aplicación.

Cuando lanzamos una operación en un hilo separado, la aplicación podrá seguir funcionando de forma normal. En dependencia de la tarea en cuestión permitiremos que el usuario siga interactuando con la app y mostraremos un aviso de “loading” para informarle de que hay una tarea en curso y que tiene que esperar para ver el recurso que ha solicitado.

En esta práctica veremos cómo crear hilos de tres formas diferentes:

* Mediante la clase **Thread**. Forma heredada de Java.
* Mediante la clase **AsyncTask**. Que ya está obsoleta pero que vamos a aprender a utilizar por haber sido muy utilizada durante años.
* Mediante **corrutinas**, que son las nuevas utilidades de concurrencia de Kotlin.

## Thread

Un hilo es un flujo de control dentro de un programa, que nos permite realizar varias tareas de forma paralela. En Kotlin y Java los hilos se crean mediante la clase Thread, para lo que contamos con dos opciones: crear una clase que herede de Thread redefiniendo su método run() o implementar la interfaz Runnable, que nos obliga a definir el método run(). A continuación se incluye un ejemplo de estas dos opciones:

class EjemploThread : Thread() {

override fun run(){

//Código del hilo

}

}

class EjemploRunnable : Runnable {

override fun run(){

//Código del hilo

}

}

En ambos casos definiremos el método que es el que se llamará al iniciar el hilo, el que deberá contener el código a ejecutar y el que finalizará el hilo al terminar su ejecución.

Después de definir la clase de nuestro hilo, lo instanciaremos y ejecutaremos llamando a su método start. Para los ejemplos anteriores:

val t1 = EjemploThread()

t1.start()

val t2 = Thread(EjemploRunnable())

t2.start()

### Acceso a la interfaz gráfica

No podemos acceder a ningún elemento de la interfaz gráfica desde dentro de un hilo. Si quisiéramos hacerlo:

* Podríamos utilizar el método View.post(Runnable) de la vista.
* Podríamos llamar runOnUiThread(Runnable) para ejecutar un bloque de código en el hilo de la interfaz.

Ambas soluciones genera un código que puede llegar a ser bastante complejo si tenemos que realizar bucles, accesos a la interfaz de forma iterativa,…

## AsyncTask

“Deprecated” desde **API30**, pero muy utilizada.

La estructura genérica para definir una tarea utilizando un AsyncTask es la siguiente:

class MiTarea : AsyncTask<ENTRADA?, PROGRESO?, SALIDA>() {

override fun **onPreExecute**(){

//…

}

override fun **doInBackground**(vararg params: ENTRADA) : SALIDA {

//…

**publishProgress**(PROGRESO)

//…

return SALIDA

}

override fun **onProgressUpdate**(vararg params: PROGRESO) {

//…

}

override fun **onPostExecute**(result: SALIDA) {

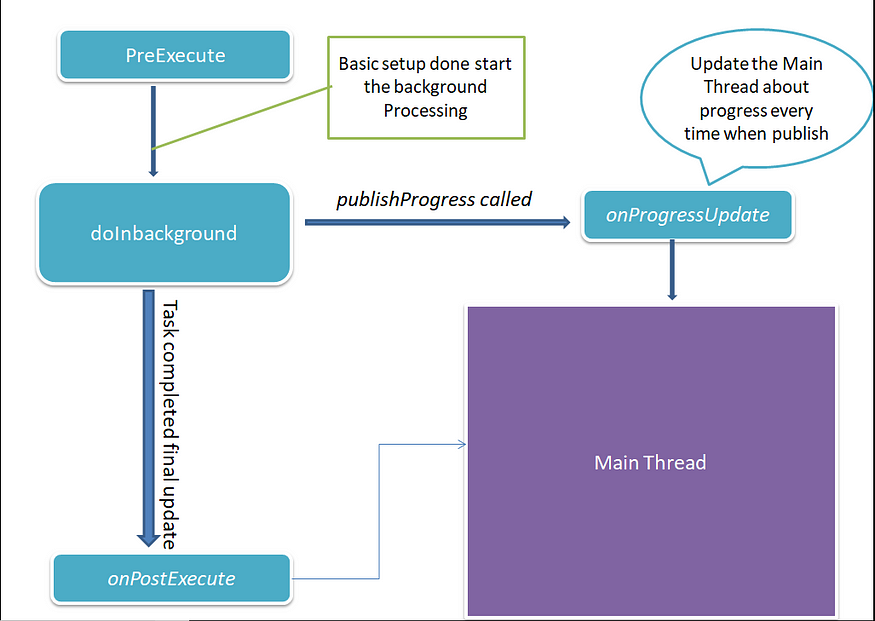
//…

}

override fun **onCancelled**(){

//…

}



El único método que se ejecuta en el hilo secundario es **doInBackground**. El resto de métodos utilizan el mismo hilo que la interfaz gráfica. Por lo tanto, cualquier cambio en la interfaz lo llevaremos a cabo desde alguno de los otros métodos, reservando doInBackground exclusivamente para la tarea asíncrona.

En la definición de **AsyncTask** se especifican tres tipos de datos:

* El primero es el tipo de dato que recibe el método doInBackground como entrada. La notación varargs params:ENTRADA, significa que puede recibir un número indeterminado de parámetros del tipo especificado. La función en realidad recibe un array de valores, por lo que accederemos a ellos con params[0], params[1], etc., y obtendremos el tamaño del array con params.size.
* El segundo tipo de dato (PROGRESO) es el que sirve para comunicar el avance de la tarea. Si estamos realizando un proceso que va a durar bastante, contamos con la posibilidad de publicar actualizaciones visuales del progreso realizado. Como ya sabemos, esto no se puede hacer directamente desde el hilo secundario (método doInBackground), pero sí podemos llamar a **publishProgress** y pasarle la información que deseamos brindar. Al llamar a dicho método se ejecutará **onProgressUpdate**, que recibirá la información que le hemos pasado como parámetro. Este método sí que se ejecuta dentro del hilo de la interfaz, por lo que es posible actualizar la visualización del progreso desde él. Es importante entender que la ejecución de onProgressUpdate no tiene por qué ocurrir inmediatamente después de la petición publishProgress, y puede incluso no llegar a ocurrir nunca si la ejecución finaliza antes.
* El tercer tipo (SALIDA) corresponde al resultado de la operación. Es el tipo de datos que devolverá doInBackground tras ejecutarse y el que recibirá **onPostExecute** como parámetro. Este último método podrá actualizar la interfaz con la información resultante de la ejecución en segundo plano.

También contamos con el método **onPreExecute**, que se ejecuta justo antes de comenzar la tarea en segundo plano, y con **onCancelled**, que se llama solamente si la tarea es cancelada (una tarea se cancela llamando a su método cancel y, en tal caso, no llegará a ejecutarse onPostExecute)

### Ejemplo de aplicación que utiliza AsyncTask

Ver el ejemplo que he subido al aula virtual. Vamos a tratar de entenderlo.

Es muy sencillo. Simplemente al pulsar el botón se ejecuta un subproceso que lo único que hace es contar los parámetros que se le pasan como entrada para ir mostrándolos en el log del sistema.

## Corrutinas

Debido a varios inconvenientes de las AsyncTask, como un **comportamiento inconsistente** dependiendo de versiones de la plataforma, ocultación de excepciones en el hilo secundario, pérdidas de contexto o pérdidas de respuestas, esta clase se ha marcado como “deprecated” desde el API 30.

Las corrutinas son un patrón de diseño de simultaneidad desarrollado por Kotlin que se pueden usar en Android para simplificar la implementación de código asíncrono. Además, aportan varias ventajas, como una **ejecución menos pesada**, lo que permite ejecutar muchas corrutinas en un mismo subproceso, **menos fugas de memoria**, se pueden suspender y cancelar, y están integradas con los últimos paquetes de Jetpack.

Las corrutinas añaden el concepto de “concurrencia estructurada” donde:

* Cada corrutina necesita iniciarse en un ámbito lógico y con un limitado ciclo de vida.
* Corrutinas iniciadas en el mismo ámbito forman una jerarquía.
* Un “job” padre no terminará hasta que todos sus hijos terminen.
* Cancelar a un padre cancelará a todos los hijos. Cancelar a un hijo no cancelará necesariamente al padre o hermanos.
* Si una corrutina hijo falla, la ejecución se propaga hacia arriba y todos los hermanos se pueden llegar a cancelar (dependiendo del tipo de “job”).

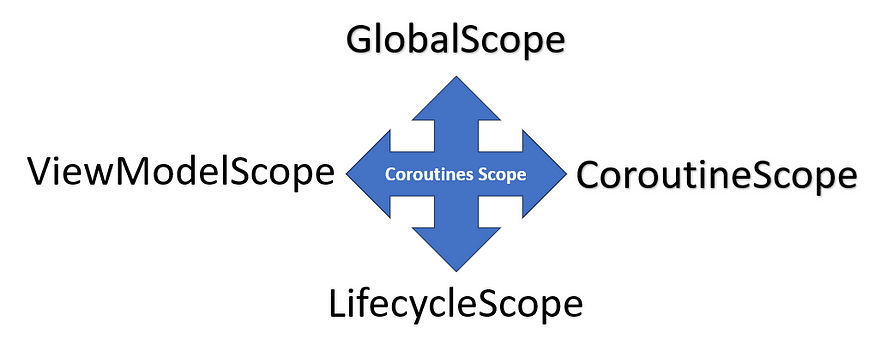
Para usar corrutinas en tu proyecto de Android, agrega la siguiente **dependencia** al archivo build.gradle de tu app:

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

Existen diferentes formas de trabajar con corrutinas:

### Coroutine Scope

Define el contexto/ámbito en el cual la “coroutine” se lanza y gestiona su ciclo de vida. Es muy importante para la gestión de la ejecución y evitar fugas de memoria.



### CoroutineScope

Es un ámbito personalizado que defines en tu clase o componente. Ayuda a gestionar “coroutines” dentro de un contexto específico y asegura poder cancelarlas cuando ya no se necesitan.

Ejemplo:

|  |
| --- |
| class MyClass {  private val scope = **CoroutineScope(Dispatchers.Default)**   fun startWork() {  scope.launch {  delay(1000)  println("Coroutine in custom scope completed")  }  }   fun cancelWork() {  scope.cancel() // Cancel all coroutines in this scope  } }  fun main() {  val myClass = MyClass()  myClass.startWork()   Thread.sleep(2000)   myClass.cancelWork() // Cancel any ongoing work }  **Salida:**  > Coroutine in custom scope completed |

### GlobalScope

GlobalScope es una instancia especial:

* De CoroutineScope que no está asociado a ningún ciclo de vida o contexto específico.
* Que ocupa un ámbito “top-level” que se extiende por toda la aplicación.
* Se utiliza cuando su ciclo de vida va más allá de un Activity o Fragment específico.

Contras:

* No es consciente del ciclo de vida 🡪 En caso de cancelaciones puede llevarnos a pérdida de memoria y comportamientos inesperados.
* Pérdida de recursos si la corrutina existe más allá del componente que la creó.

Ejemplo:

|  |
| --- |
| fun main() {  // Start a coroutine in the global scope  **GlobalScope.launch** {  delay(1000)  println("GlobalScope coroutine completed")  }   println("Main thread is free")   Thread.sleep(2000) }  **Salida:**  > Main thread is free > GlobalScope coroutine completed |

### ViewModelScope

Es la CoroutineScope predefinida por Android. Está diseñada para ser utilizada con ViewModels.

ViewModelScope se cancela automáticamente cuando su ViewModel asociado se destruye.

### Tipos de dispatcher

En el método “launch” se especifica el dispatcher (emisor o tipo de hilo) donde tiene que ejecutar el código. Kotlin proporciona tres:

* **Dispatcher.Main**: Si el hilo va a modificar/actualizar datos en el “padre” utilizaremos este dispatcher ya que de forma natural podemos tratar sus vistas.
* **Dispatcher.Default**: Si el hilo va a realizar un uso intensivo de CPU utilizaremos este dispatcher. Esto es debido a que tiene un número limitado de threads (lo marca el número de núcleos del procesador). Potencial bloqueo en caso de utilizarlo para múltiples llamadas de entrada/salida.
* **Dispatcher.IO**: Si el hilo va a utilizar muchas acciones de entrada salida ya que este dispatcher ofrece un número dinámico de threads independientemente del número de núcleos del procesador.

### Ejemplo de uso

Se va a crear una corrutina que lance un log al inicio y otro al final de un hilo.

Clase MainViewModel

|  |
| --- |
| package com.example.probandoviewmodelscope  import android.util.Log import androidx.lifecycle.ViewModel import kotlinx.coroutines.CoroutineScope import kotlinx.coroutines.Dispatchers import kotlinx.coroutines.SupervisorJob import kotlinx.coroutines.cancel import kotlinx.coroutines.delay import kotlinx.coroutines.launch import kotlin.coroutines.cancellation.CancellationException  class MainViewModel : ViewModel() {   *// CoroutineScope associated with the ViewModel* private val viewModelScope = *CoroutineScope*(Dispatchers.Main + *SupervisorJob*())   fun performBackgroundTask() {  *// Launch a coroutine within the ViewModel scope* viewModelScope.*launch* **{** try {  Log.i("msg","Background task ini.")  *// Simulate a background task* delay(2000)  Log.i("msg","Background task completed")  *println*("Background task completed")  } catch (e: CancellationException) {  *// Handle cancellation if needed  println*("Coroutine was canceled")  }  **}** }   override fun onCleared() {  super.onCleared()  *// Cancel all coroutines when the ViewModel is cleared (e.g., when the associated UI component is destroyed)* viewModelScope.*cancel*()  } } |

Clase principal

|  |
| --- |
| …  var hilo:MainViewModel = MainViewModel()  hilo.performBackgroundTask()  … |

**Nota sobre SupervisorJob()**:

<https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/-supervisor-job.html>

### Livedata

Livedata es una clase que representa una envoltura para datos que deseas observar dentro de un ciclo de vida.

Esta clase **empareja un objeto** LifecycleOwner **con otro** Observer a fin de que el observer notifique datos solo si el owner está en estado activo (Lifecycle.State.STARTED y Lifecycle.State.RESUMED).

De esta forma, al usar LiveData podremos:

* Evitar leaks de memoria al dejar de recibir cambios de datos cuando el ciclo de vida se encuentra en inactividad
* Mantener la UI actualizada al cambiar los datos (la vista recibe actualizaciones de estado en vez de solicitarlas)
* Recibe los datos más nuevos incluso en cambios de configuración

Como Usar Objetos LiveData:

### Actividad actualizando UI

Vamos a crear una app que contenga dos botones. En uno de ellos llamaremos a un método que escriba un mensaje en un TextView antes y después de ejecutarse (Delay de 10 segundos). El otro escribirá el mensaje directamente.

Para probar que funciona, pulsaremos el primer botón y luego el segundo. Observa que aunque la tarea del primer botón tarda 10 segundos en ejecutarse, me permite pulsar el segundo botón y reflejar lo realizado en el TextView. Pasados los 10 segundos se actualizará el textView con el mensaje lanzado por el subproceso.

Activity\_main.xml

|  |
| --- |
| *<?*xml version="1.0" encoding="utf-8"*?>* <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:orientation="vertical"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent">   <Button  android:id="@+id/btnHilo"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content"  android:text="tarea en hilo" />   <Button  android:id="@+id/btnInmediato"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content"  android:text="Tarea inmediata" />   <TextView  android:id="@+id/lblResultado"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content"  android:text="TODO" /> </LinearLayout> |

MainActivity

|  |
| --- |
| package com.example.probandoviewmodelscope  import android.os.Bundle import android.widget.Button import android.widget.TextView import androidx.activity.ComponentActivity import androidx.activity.compose.setContent import androidx.activity.enableEdgeToEdge import androidx.activity.viewModels import androidx.appcompat.app.AppCompatActivity import androidx.compose.foundation.layout.fillMaxSize import androidx.compose.foundation.layout.padding import androidx.compose.material3.Scaffold import androidx.compose.material3.Text import androidx.compose.runtime.Composable import androidx.compose.ui.Modifier import androidx.compose.ui.tooling.preview.Preview import com.example.probandoviewmodelscope.ui.theme.ProbandoViewModelScopeTheme  class MainActivity : AppCompatActivity() {  // Kotlin delegate (by viewModels())  // Inicializa el viewModel  private val viewModel: MainViewModel by *viewModels*()   override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  *//enableEdgeToEdge()* setContentView(R.layout.*activity\_main*)   val bHilo = findViewById<Button>(R.id.*btnHilo*)  val bInmediato = findViewById<Button>(R.id.*btnInmediato*)  val texto = findViewById<TextView>(R.id.*lblResultado*)   bHilo.setOnClickListener **{** *// Observa los cambios en el LiveData* viewModel.text.observe(this) **{** newText **->** texto.*text* = newText  **}** viewModel.performBackgroundTask()   **}** bInmediato.setOnClickListener **{** texto.*text* = "Tarea inmediata finalizada"  **}** } } |

**Nota**:

<https://medium.com/@yadavshashank700/a-simple-view-of-viewmodels-61c249d1bd6f>

MainViewModel

|  |
| --- |
| package com.example.probandoviewmodelscope  import android.util.Log import android.widget.TextView import androidx.lifecycle.LiveData import androidx.lifecycle.MutableLiveData import androidx.lifecycle.ViewModel import kotlinx.coroutines.CoroutineScope import kotlinx.coroutines.Dispatchers import kotlinx.coroutines.SupervisorJob import kotlinx.coroutines.cancel import kotlinx.coroutines.delay import kotlinx.coroutines.launch import kotlin.coroutines.cancellation.CancellationException  class MainViewModel : ViewModel() {   *// CoroutineScope associated with the ViewModel* private val viewModelScope = *CoroutineScope*(Dispatchers.Main + *SupervisorJob*())   private val \_text = MutableLiveData<String>()  val text: LiveData<String> get() = \_text   fun performBackgroundTask() {  *// Launch a coroutine within the ViewModel scope* viewModelScope.*launch* **{** try {  \_text.postValue("Tarea en hilo comenzada") *// Actualiza el LiveData* delay(10000)  \_text.postValue("Tarea en hilo Finalizada") *// Actualiza el LiveData* } catch (e: CancellationException) {  *// Handle cancellation if needed  println*("Coroutine was canceled")  }  **}** }   override fun onCleared() {  super.onCleared()  *// Cancel all coroutines when the ViewModel is cleared (e.g., when the associated UI component is destroyed)* viewModelScope.*cancel*()  } } |

# Webgrafía

<https://medium.com/nplix/android-asynctask-example-in-kotlin-for-background-processing-of-task-59ed88d8c545>

<https://medium.com/@paritasampa95/coroutinescope-coroutinecontext-discussed-in-depth-to-understand-better-part-2-550dbc7af3d2>

<https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471>

<https://www.develou.com/lifecycle/>