{desafío} latam_

Threads y Coroutines (Parte II)

AsyncTask

Competencias:

- Conocer la clase AsyncTask de android
- Implementar tareas con la clase AsyncTask de android en un proyecto

Introducción

En este capítulo estudiaremos la clase AsyncTask de android para el desarrollo de tareas asíncronas que permiten el envío de una petición, continuando con el flujo normal del sistema, para posteriormente recibir la respuesta a dicha petición y ser utilizada, por ejemplo, en forma de datos dentro de nuestra aplicación.

La clase AsyncTask es ampliamente utilizada en muchas aplicaciones android desarrolladas en la actualidad, además de ser una clase con un continuo soporte y desarrollo por parte del equipo de android.

AsyncTask realiza operaciones en el subproceso en segundo plano (background thread) y se actualizará en el subproceso principal (thread UI). Asynctask nos ayuda a establecer comunicación entre hilos secundarios y principales (subThreads y mainThread/threadUI).

La clase AsyncTask

AsyncTask permite el uso correcto y fácil del main thread UI de android. Esta clase nos ayuda a realizar operaciones en segundo plano y publicar resultados en el subproceso de la UI (interfaz de usuario) sin tener que manipular subprocesos, controladores o servicios.

AsyncTask está diseñado para ser una clase de ayuda en todo Thread y Handler, y no constituye un framework genérico para threads. Las AsyncTasks deben usarse idealmente para operaciones cortas, de unos pocos segundos.

Si se desea mantener los subprocesos en ejecución durante largos períodos de tiempo, es recomendable utilizar las diversas herramientas del API proporcionadas por android, como lo son los servicios, la clase Executor, ThreadPoolExecutor y, entre otras.

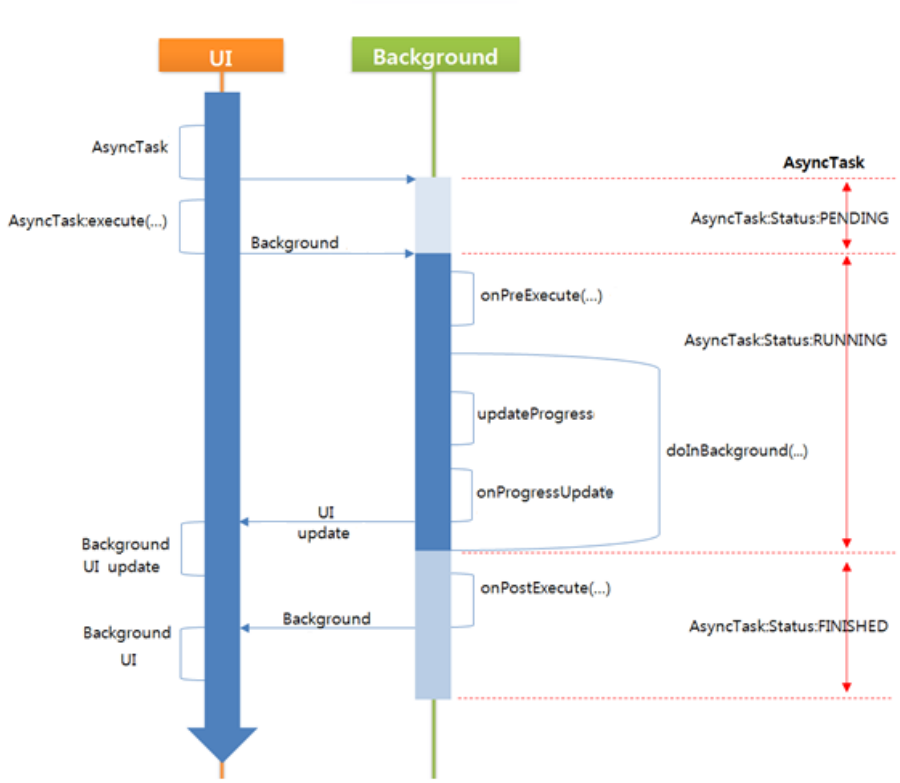

Imagen 1. Diagrama proceso AsyncTask UI Thread y Background Thread.

Una AsyncTask se define mediante tareas que se ejecuta en un subproceso en segundo plano, y cuyo resultado se publica en el subproceso de la interfaz de usuario. Un AsyncTask define tres métodos genéricos:

- doInBackground(), para todo aquel código que se ejecutará en segundo plano.
- onProgressUpdate(), que recibe toda actualización del progreso del método doInBackground.
- onPostExecute(), que lo utilizamos para actualizar la interfaz de usuario (UI) una vez el proceso en segundo plano se haya completado.

Una clase auxiliar AsyncTask se vería de la siguiente forma:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
protected Long doInBackground(URL... urls) {
// code that will run in the background
return ;
}

protected void onProgressUpdate(Integer... progress) {
// receive progress updates from doInBackground
}

protected void onPostExecute(Long result) {
// update the UI after background processes completes
}
}
```

Iniciar un AsyncTask

Una clase AsyncTask se ejecutaría desde el mainThread a través de una actividad o fragmento de la siguiente forma:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Cancelar un AsyncTask

Una tarea puede cancelarse en cualquier momento invocando el método cancel(). Invocar este método hará que las siguientes llamadas (calls) invoquen al método isCancelled(), y este retornará el valor booleano true.

Una vez invocado el método cancel() la aplicación en vez de ejecutar el método onPostExecute() de AsyncTask ejecutará este método isCancelled(). Para asegurarnos de cancelar la tarea lo más rápido posible es oportuno validar el retorno de este método isCancelled() durante la ejecución del método doInBackGround().

Indicaciones para el uso correcto de AsyncTask

- La clase AsyncTask debe cargarse en el hilo de la interfaz de usuario (UI Thread). Esto se hace automáticamente a partir de la versión android JELLY_BEAN.
- La instancia de la tarea debe crearse en el subproceso de la interfaz de usuario (UI Thread).
- El método execute, debe invocarse en el hilo de la interfaz de usuario (UI Thread).
- No se debe invocar los métodos de la clase AsyncTask por separado o manualmente doInBackGround, PostExecute, etc.).
- La tarea se puede ejecutar solo una vez. Si se intentará ejecutar por una segunda vez una excepción debe ser enviada.

El método onPreExecute de AsyncTask

Este método se puede invocar en el subproceso de la interfaz de usuario antes de ejecutar la tarea (UI Thread). Este paso se usa normalmente para configurar la tarea, por ejemplo, iniciando una barra de progreso en la interfaz de usuario.

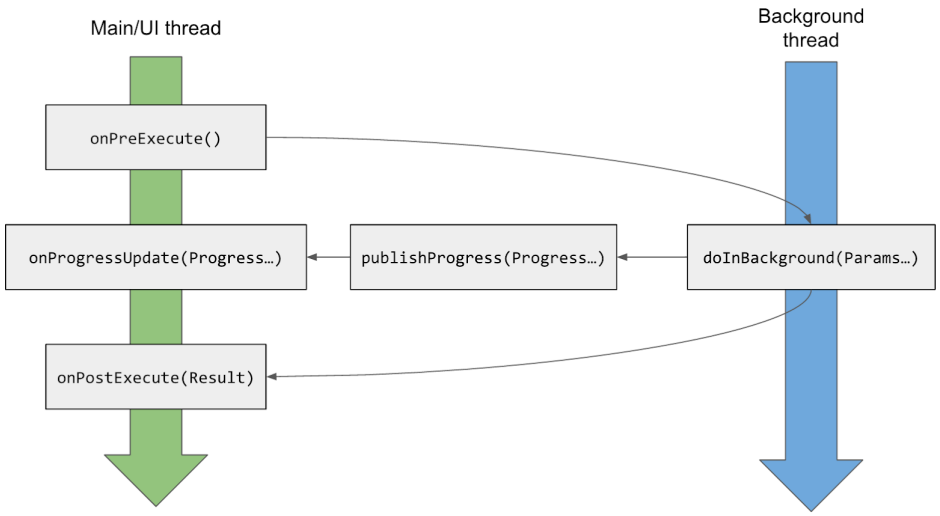

Imagen 2. Diagrama simple AsyncTask Threads.

Ejercicio 5:

Crear un proyecto llamado “AsyncTask” donde se descargue al momento de iniciarse la aplicación, la imagen del día de la nasa en formato HD y sea instanciada la imagen en una vista tipo ImageView. También se requiere el uso de una progressbar que se muestre en el lugar que ocupará la imagen una vez se haya descargado.

1. Editamos nuestro archivo de diseño “activity_main.xml” para incluir tres vistas, un textview, un progressbar y un imageview de la siguiente forma:

```
<TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="AsyncTask Download Image Example"
      app:layout_constraintLeft_toLeftOf="parent"
      app:layout_constraintRight_toRightOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      android:layout_marginTop="100dp"
      android:textSize="20dp"
      android:textAlignment="center"/>

<ProgressBar android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:id="@+id/progressBar"
          style="?android:attr/progressBarStyleLarge"
          app:layout_constraintLeft_toLeftOf="parent"
          app:layout_constraintRight_toRightOf="parent"
          app:layout_constraintBottom_toBottomOf="parent"
          app:layout_constraintTop_toTopOf="parent"
          android:visibility="gone"
/>

<ImageView android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:id="@+id/imgOfDay"
         app:layout_constraintLeft_toLeftOf="parent"
         app:layout_constraintRight_toRightOf="parent"
         app:layout_constraintBottom_toBottomOf="parent"
         app:layout_constraintTop_toTopOf="parent"
         android:contentDescription="Nasa image of day"
/>
```

2. En nuestro método onCreate() de la “MainActivity”, iniciamos nuestros objetos de vista de la siguiente manera:

```
val imgOfDay = findViewById<ImageView>(R.id.imgOfDay)
val progressBar = findViewById<ProgressBar>(R.id.progressBar)
```

3. Dentro de nuestra actividad, crearemos una subclase privada llamada DownloadTask, que reciba como parámetros en su constructor las variables imgOfDat y progressBar, además de recibir la url de la imagen del día dentro de sus parámetros propios AsyncTask, y declarando el tipo de retorno como un bitmap para ser manipulado en el método postExecute(). Por otra parte, en nuestro método doInBackground debemos hacer visible la progressBar y programar la descarga de la imagen haciendo uso de java.net.URL y BitmapFactory, como hemos visto en ejemplos anteriores. El código es de la siguiente manera:

```
private class DownloadImageTask(val imgOfDay: ImageView, val progressBar: ProgressBar) :
AsyncTask<String, Void, Bitmap>() {

    override fun doInBackground(vararg urls: String): Bitmap? {
        val urldisplay = urls[0]
        var bmp: Bitmap? = null
        try {
            progressBar.visibility = View.VISIBLE

            val inputStream = java.net.URL(urldisplay).openStream()
            bmp = BitmapFactory.decodeStream(inputStream)
        } catch (e: Exception) {
            Log.e("Error", e.message)
            e.printStackTrace()
        }

        return bmp
    }

    override fun onPostExecute(result: Bitmap?) {
        try {
            progressBar.visibility = View.GONE
            imgOfDay.setImageBitmap(result)
        }catch (e: java.lang.Exception){e.printStackTrace()}
    }
```

4. Por último, al final de nuestro método onCreate() debemos declarar el llamado a esta nueva clase DownloadTask de la siguiente manera:

```
DownloadImageTask(imgOfDay, progressBar).execute("https://apod.nasa.gov/apod/image/1908/M61-
HST-ESO-S1024.jpg")
```

5. Para finalizar no se nos escape agregar los permisos necesarios a nuestro archivo manifest:

```
<uses-permission android:name="android.permission.INTERNET" />
```

6. Al ejecutar la aplicación observaremos el inicio del trabajo y su fin, como lo muestran las siguientes imágenes:

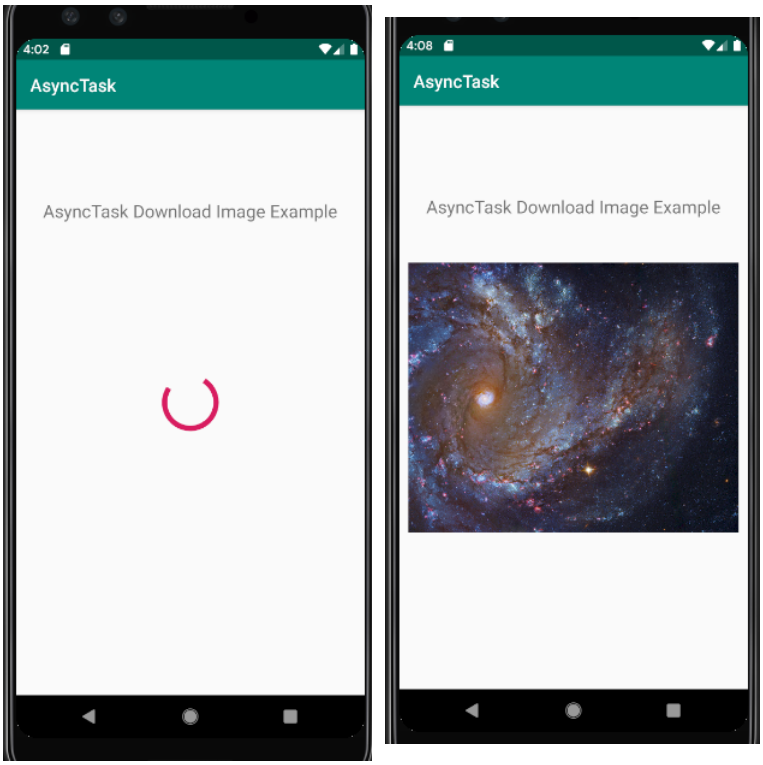

Imagen 3. AsyncTask Download.

Promises y Coroutines

Competencias:

- Conocer el patrón moderno de desarrollo de promesas para tareas asíncronas.
- Aprender sobre las coroutines y sus conceptos modernos en android.
- Entender la asociación entre promesas y coroutines
- Implementar promesas y coroutines en un proyecto android.

Introducción

En este capítulo estudiaremos el concepto moderno de las promesas y coroutines, su evolución y utilización para la ejecución de tareas y procesos asíncronos.

Tanto las promesas como las coroutines se pueden interpretar como una forma de escribir código asíncrono de manera moderna y eficiente, ambas surgen de la evolución de antiguos procesos y formas de realizar procesos asíncronos.

Es importante aprender a trabajar con promesas y coroutines en android para en primer lugar ahorrarnos grandes dolores de cabeza por errores de agrupación de llamados ya superados, y también para escribir un código más limpio, eficiente y fácil de leer.

Promises

Pensemos en una promesa como un objeto que espera a que finalice una acción asincrónica, luego llama a una segunda función. Podemos programar esa segunda función llamando al método .then() y pasando la nueva función. Cuando finaliza esta última función asincrónica, da su resultado a la promesa y la promesa lo da a la siguiente función (como parámetro).

Cada llamada al método .then() espera la promesa anterior y ejecuta la siguiente función, luego convierte el resultado en una promesa si es necesario. Esto nos permite encadenar sin problemas llamadas síncronas y asíncronas. Simplifica tanto el código, que la mayoría de las nuevas especificaciones devuelven promesas de sus métodos asincrónicos.

Las promesas ofrecen una mejor manera de trabajar con el código y llamados asíncronos, estando presentes en el mercado por un tiempo considerable a través de distintas librerías.

Las promesas en principio surgen como iniciativa para reemplazar los callbacks, motivado a sus fallas o agrupaciones interminables (callback hell) que no ayudaban en mucho a la lectura del código del sistema, y a su vez creaban fallas difíciles de seguir o debuggear. Las promesas han ido evolucionando con el tiempo dando origen a las nuevas versiones de async/await de ES2017 (EcmaScript).

CallBack vs Promises

El siguiente es un ejemplo de un método que valida si un usuario es muy joven según su edad utilizando callbacks:

```
fun isUserTooYoung(id, callback) {
  openDatabase(fun(db) {
    getCollection(db, 'users', fun(col) {
      find(col, {'id': id}, fun(result) {
        result.filter(fun(user) {
          callback(user.age < cutoffAge);
        });
      });
    });
  });
}
```

De este código podemos apreciar en primera instancia la cantidad de llaves de cierre al final, lo que fácilmente terminaría en un error de compilación por la falta de alguna de estas, sin imaginar la tortura que sería agregar una condición más a esta función.

Manteniendo la misma validación anterior de si un usuario es muy jóven para acceder, veamos cómo quedaría el código utilizando las promesas:

```
fun isUserTooYoung(id): Db {
  return openDatabase() // returns a promise
  .then(fun(db: Database) {return getCollection(db, 'users');})
  .then(fun(col: Collection) {return find(col, {'id': id});})
  .then(fun(user: User) {return user.age < cutoffAge;});
}
```

A primeras, se entiende que es un código más legible y tiene tres acciones subsecuentes.

Promises.All

A menudo queremos ejecutar acciones sólo después de que una colección de operaciones asincrónicas se haya completado con éxito. El objeto Promise.all devuelve una promesa que se resuelve si todas las promesas pasadas se resuelven. Si alguna de las promesas aprobadas rechaza, Promise.all rechaza con la razón de la primera promesa que rechazó. Esto es muy útil para garantizar que se complete un grupo de acciones asincrónicas antes de continuar con otro paso.

En el ejemplo a continuación, promise1 y promise2 devuelven promesas. Queremos que ambas se carguen antes de continuar. Ambas promesas pasan como parámetros al objeto Promise.all. Si cualquiera de las solicitudes es rechazada, Promise.all rechaza toda la acción con el resultado de la promesa rechazada. Si se cumplen ambas solicitudes, Promise.all resuelve con los valores de ambas promesas (como una lista).

```
var promise1 = getJSON('/users.json');
var promise2 = getJSON('/articles.json');

Promise.all([promise1, promise2]) // Array of promises to complete
.then(fun(results) {
  println('all data has loaded');
})
.catch(fun(error) {
  println('one or more requests have failed: ' + error);
});
```

Coroutines

Una corrutina es un patrón de diseño de concurrencia que puede usarse en Android para simplificar el código que se ejecuta de forma asincrónica. Se agregaron corutinas a Kotlin en la versión 1.3 y se basan en conceptos establecidos de otros idiomas.

Las corrutinas podrían interpretarse como la última versión, la más avanzada y moderna de los callbacks y promesas para android, ya que en realidad las corrutinas, son en mucho el patrón async/await de javascript que lidera este avance en operaciones asíncronas.

En Android, las rutinas ayudan a resolver dos problemas principales:

- Administrar tareas de larga duración que de otro modo podrían bloquear el hilo principal y hacer que su aplicación se congele.
- Proporcionar seguridad principal, o llamadas con seguridad a las operaciones de red o disco desde el hilo principal.

Coroutines en tareas extensas

En Android, cada aplicación tiene un hilo principal que maneja la interfaz de usuario y gestiona las interacciones del usuario. Si una aplicación está asignando demasiado trabajo al hilo principal, la aplicación puede congelarse o detenerse. Las solicitudes de datos a través de grandes apis, el análisis de objetos JSON extensos, la lectura o la escritura desde una base de datos, o incluso la iteración de listas grandes, pueden hacer que la aplicación se ejecute lo suficientemente lento como para provocar que nuestra pantalla responda lentamente a eventos táctiles. Estas operaciones de larga duración deben ejecutarse fuera del hilo principal.

El siguiente ejemplo muestra la implementación de rutinas simples para una tarea hipotética de larga duración:

```
suspend fun fetchDocs() {                      // Dispatchers.Main
    val result = get("https://developer.android.com") // Dispatchers.IO for `get`
    show(result)                               // Dispatchers.Main
}

suspend fun get(url: String) = withContext(Dispatchers.IO) { /* ... */ }
```

Las corrutinas se basan en funciones regulares agregando dos operaciones para manejar tareas de larga duración. Además de invoke(o call) y return, las corrutinas agregan suspend y resume:

- **suspend**: hace pausa en la ejecución de la rutina actual, guardando todas las variables locales.
- **resume:** continúa la ejecución de una corrutina suspendida desde el lugar donde fue suspendida.

En el ejemplo anterior, el método get() todavía se ejecuta en el hilo principal, pero suspende la rutina antes de que comience la solicitud de red.

Cuando finaliza la solicitud de red, get() reanuda la rutina suspendida en lugar de utilizar una devolución de llamada para notificar al hilo principal.

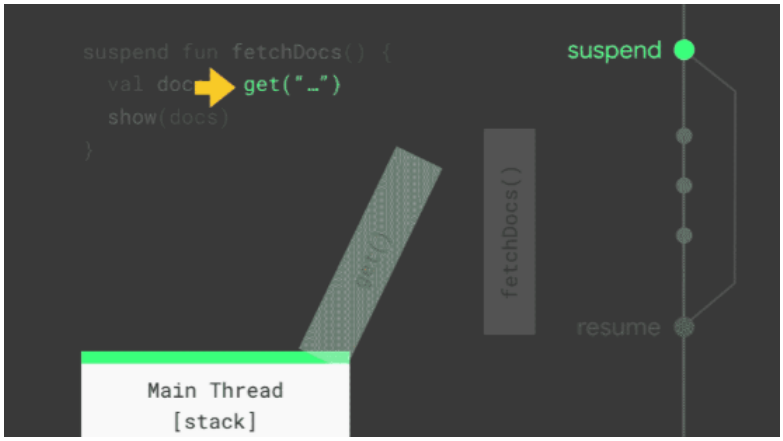

Imagen 4. Funcionamiento suspend resume en coroutines.

Imagen 5. Funcionamiento Dispatcher.IO resume en coroutines.

Imagen 6. Funcionamiento resume en coroutines.

Seguridad y Coroutines

Las corrutinas de Kotlin usan despachadores para determinar qué hilos se usan para la ejecución de la rutina. Para ejecutar el código fuera del hilo principal, puede indicarle a las corrutinas de Kotlin que realicen el trabajo en un despachador predeterminado. En Kotlin, todas las corrutinas deben ejecutarse en un despachador, incluso cuando se ejecutan en el hilo principal. Las corrutinas pueden suspenderse, y el despachador es responsable de reanudarlas.

Para especificar dónde deben ejecutarse las corrutinas, Kotlin proporciona tres despachadores que puede usar:

- **Dispatchers.Main**: este despachador se usa para ejecutar una rutina en el hilo principal de Android. Este debe usarse solo para interactuar con la interfaz de usuario y realizar un trabajo rápido.
- **Dispatchers.IO**: este despachador está optimizado para realizar operaciones de disco o de red fuera del hilo principal.
- **Dispatchers.Default**: este despachador está optimizado para realizar un trabajo intensivo de CPU fuera del hilo principal. Un ejemplo podría ser ordenar una lista y analizar un objeto JSON.

```
suspend fun fetchDocs() {                    // Dispatchers.Main
    val result = get("developer.android.com")  // Dispatchers.Main
    show(result)                             // Dispatchers.Main
}

suspend fun get(url: String) =               // Dispatchers.Main
    withContext(Dispatchers.IO) {            // Dispatchers.IO (main-safety block)
        /* perform network IO here */        // Dispatchers.IO (main-safety block)
    }                                        // Dispatchers.Main
}
```

Iniciando Coroutines

Podemos comenzar las corrutinas de una de dos maneras:

- **launch**: comienza una nueva corrutina y no devuelve el resultado al objeto que llama. Cualquier trabajo que se considere iniciarse y no esperar por su respuesta es ideal indicar la palabra launch.
- **async**: inicia una nueva rutina y le permite devolver un resultado con una función de suspensión llamada await.

Por lo general, ejecutamos launch para iniciar una nueva corrutina de una función regular, ya que una función regular no puede llamar await. Para usar async debemos hacerlo sólo cuando se esté dentro de otra corrutina o cuando se esté dentro de una función de suspensión.

```
fun onDocsNeeded() {
    viewModelScope.launch {    // Dispatchers.Main
        fetchDocs()            // Dispatchers.Main (suspend function call)
    }
}
```

Uso de Coroutines en Android

Para utilizar las corrutinas en nuestros proyectos android es necesario implementar en el gradle de la app las librerías que nos permiten tener acceso a estas. Por ejemplo el siguiente código de versión 1.0.1:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.1'
```

Ejercicio 6:

Crear un proyecto llamado “Coroutines” que realice la descarga de la imagen del día de la nasa y la muestre en un imageview, utilizando java.net.URL. La actividad debe implementar CoroutineScope y tener un método que aplique launch{}, que a su vez llame otro método que ejecuta la descarga utilizando una referencia a Dispatchers.IO.

1. Agregamos la librería de coroutines en nuestro archivo gradle de la carpeta app.

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.1'
```

2. Copiamos las vistas creadas en el ejercicio del capítulo “AsyncTask” en nuestro archivo layout activity_main.xml de la siguiente manera:

```
<TextView
      android:id="@+id/title"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Coroutine Download Image Example"
      app:layout_constraintLeft_toLeftOf="parent"
      app:layout_constraintRight_toRightOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      android:layout_marginTop="100dp"
      android:textSize="20dp"
      android:textAlignment="center"/>

<ProgressBar android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:id="@+id/progressBar"
          style="?android:attr/progressBarStyleLarge"
          app:layout_constraintLeft_toLeftOf="parent"
          app:layout_constraintRight_toRightOf="parent"
          app:layout_constraintBottom_toBottomOf="parent"
          app:layout_constraintTop_toTopOf="parent"
          android:visibility="gone"
/>

<ImageView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/imgOfDay"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:contentDescription="Nasa image of day"
/>
```

3. Extendemos nuestra clase actividad de la clase CoroutineScope, reescribiendo la variable coroutineContext de la siguiente manera:

```
class MainActivity : AppCompatActivity(), CoroutineScope {

  private val job = Job()
  override val coroutineContext = Dispatchers.Main + job
```

4. Creamos una variable de clase final para el string de nuestra imagen:

```
private val urlPictureOfTheDay: String = "https://apod.nasa.gov/apod/image/1908/M61-HST-ESO-S1024.jpg"
```

5. Creamos un método llamado “downloadImageBlocking” que ejecute la acción de descargar la imagen desde el api de la nasa y retorne un bitmap:

```
private fun downloadImageBlocking(): Bitmap? {
  val bmp: Bitmap
  try {
     val inputStream = URL(urlPictureOfTheDay).openStream()
     bmp = BitmapFactory.decodeStream(inputStream)
     return bmp

  }catch (e: Exception){
     e.printStackTrace()
  }
  return null
}
```

6. A continuación crearemos un método llamado “downloadImage” en donde aplicaremos el launch de la corrutina, habilitamos y deshabilitamos la progressBar, llamamos un subproceso Dispatchers.IO invocando el método suspend downloadImageBlocking(), para finalmente asignar el bitmap a nuestra vista imageview.

```
private fun downloadImage(){
  launch {
     progressBar.visibility = View.VISIBLE
     val bitmap: Bitmap? = withContext(Dispatchers.IO) { downloadImageBlocking() }
     if(bitmap != null) {
        imgOfDay.setImageBitmap(bitmap)
     }
     progressBar.visibility = View.GONE
  }
}
```

7. Finalmente sobreescribimos nuestro método de ciclo de vida onStart() para iniciar la corrutina:

```
override fun onStart() {
  super.onStart()
  downloadImage()
}
```

8. No olvidemos los permisos en el manifest:

```
<uses-permission android:name="android.permission.INTERNET" />
```

9. Nuestra aplicación al ejecutarla tendrá el resultado del ejercicio del capítulo AsyncTask, pero en esta ocasión el código ha sido reescrito utilizando coroutines para entender su funcionamiento y diferencia.

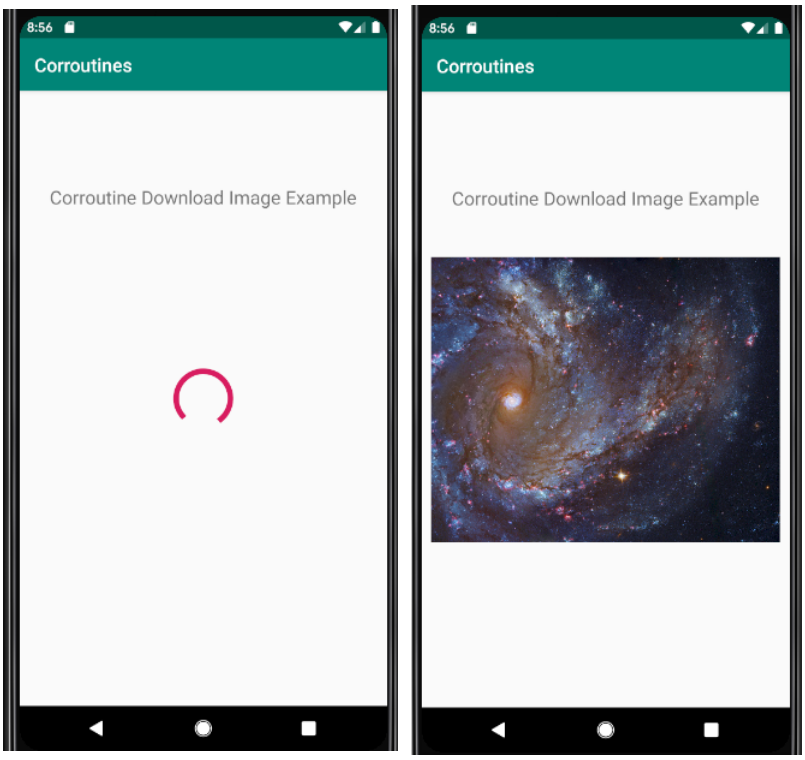

Imagen 7. Resultado del ejercicio.