Programación Multimedia y Dispositivos Móviles

UD 10. Tareas asíncronas y notificaciones

Javier Carrasco

Curso 2024 / 2025

Este obra está bajo una licencia de Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional. Última actualización: septiembre de 2023.

Tareas asíncronas y notificaciones

10. Tareas asíncronas y notificaciones 3
10.1. Hilos 3
10.2. ProgressBar 4
10.3. Tareas asíncronas 6
10.4. Corrutinas 9
10.5. Notificaciones 12
10.6. Un ejemplo útil, descarga de imágenes 15
10.6.1. Comprobar el estado de la conexión 20
10.7. Uso del vibrador 21
10.8. Reproducir un sonido 22

10. Tareas asíncronas y notificaciones

En este capítulo se tratará de poner en práctica el uso de tareas asíncronas en Android, de una manera manual, por llamarlo de alguna manera, llegando hasta el uso de corrutinas y mostrando un ejemplo "útil". Otro de los elementos que se verán serán las notificaciones en la barra de estado del sistema, elemento que permite notificar el estado de una acción, un mensaje recibido o cualquier otro evento producido.

10.1. Hilos

Los hilos son ejecuciones que se asocian a una aplicación. Están ligados a la programación concurrente, dando al usuario la sensación de multitarea. Los hilos son apropiados para tareas con una media, o alta, tasa de datos y tareas con elevado consumo de la CPU.

Figura 1

Debe tenerse en cuenta que los hilos comparten el mismo espacio de memoria, la modificación de un dato por parte de un hilo, permitirá el acceso al dato modificado por otros hilos.

Según la documentación de Android, se recomienda no acceder directamente a objetos del hilo de UI desde otros hilos. Pueden producirse anomalías debido a la ausencia de sincronización.

Observa el código del proyecto *"My Threads"*, la actividad principal contendrá cuatro botones con una etiqueta a la derecha de cada uno de ellos. La actividad principal contendrá un método que se encargará de crear un hilo. Cada uno de los botones llamará al método para iniciar un hilo, mostrando en el *TextView* correspondiente el estado de la ejecución.

```
1  private fun createThread(btn: Button, duration: Long, laps: Int, display: TextView) {
2      display.text = "0"
3      display.setBackgroundColor(Color.TRANSPARENT)
4      btn.isEnabled = false
5
6      Thread(Runnable {
7          // Se imitará la realización de una tarea.
8          var counter = 0
9
10         while (counter < laps) {
11             try {
12                 counter++
13                 Thread.sleep(duration)
14             } catch (e: InterruptedException) {
15                 e.printStackTrace()
16             }
17
18             // Permite actuar con elementos de la UI.
19             display.post { display.text = counter.toString() }
20         }
21
```

```
        // Acciones que se realizarán al finalizar la tarea.
        runOnUiThread {
            display.text = "FIN"
            display.setBackgroundColor(Color.GREEN)
            btn.isEnabled = true
            Toast.makeText(this, btn.text, Toast.LENGTH_SHORT).show()
        }
    }).start()
}
```

Las llamadas podrían ser como se muestran a continuación, cada una de ellas variando la duración de la tarea.

```
binding.btn1.setOnClickListener {
    createThread(binding.btn1, 500, 10, binding.tv1)
}
binding.btn2.setOnClickListener {
    createThread(binding.btn2, 100, 20, binding.tv2)
}
binding.btn3.setOnClickListener {
    createThread(binding.btn3, 200, 30, binding.tv3)
}
binding.btn4.setOnClickListener {
    createThread(binding.btn4, 300, 40, binding.tv4)
}
```

Puedes observar el resultado final de este código en la imagen de la figura, junto a los botones se mostrará el contador con la evolución de la tarea reproducida en cada *thread*, así como un *Toast* al finalizar cada uno de los hilos lanzados.

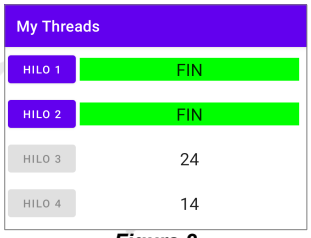

Figura 2

Para controlar que no se repita la creación de un mismo hilo, el botón se desactiva durante la ejecución, volviendo a activarse al finalizar la tarea.

10.2. ProgressBar

Las *ProgressBar* son un elemento visual que básicamente permite mostrar al usuario el progreso de una tarea en ejecución. Ahora se añadirá una *ProgressBar* (Horizontal) debajo de cada dupla *Button/TextView* en la actividad principal.

```
<ProgressBar
    android:id="@+id/progressBar1"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="5dp" />
```

Figura 3

Ahora se modificará el método `createThread` para que también muestre el progreso de la tarea mediante la *ProgressBar*.

```
private fun createThread(btn: Button,
    duration: Long,
    laps: Int,
    display: TextView,
    progress: ProgressBar
) {
    display.text = "0"
    display.setBackgroundColor(Color.TRANSPARENT)
    btn.isEnabled = false
    progress.progress = 0

    Thread(Runnable {
        // Se imitará la realización de una tarea.
        var counter = 0
        val sectionBar: Int = 100 / laps

        while (counter < laps) {
            try {
                counter++
                Thread.sleep(duration)
            } catch (e: InterruptedException) {
                e.printStackTrace()
            }

            // Permite actuar con elementos de la UI.
            display.post {
                display.text = counter.toString()
                progress.progress += sectionBar
            }
        }
        // Acciones que se realizarán al finalizar la tarea.
        runOnUiThread {
            display.text = "FIN"
            display.setBackgroundColor(Color.GREEN)
            btn.isEnabled = true
            progress.progress = 100
            Toast.makeText(this, btn.text, Toast.LENGTH_SHORT).show()
        }
    }).start()
}
```

Observa como se utiliza la propiedad `progress` del objetos *ProgressBar* para actualizar el estado de la barra. La variable *sectionBar* se utiliza para saber cuantas secciones deben hacerse de la barra para completar el progreso. Debes tener en cuenta que el progreso se hará de forma entera, la propiedad *progress* es de tipo *Int*.

10.3. Tareas asíncronas

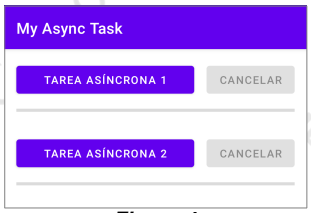

Figura 4

Las tareas asíncronas liberan al programador de la creación de hilos, sincronización y presentación de los resultados en el hilo principal. Estas tareas utilizan como sistema de procesamiento el paralelismo. Este tipo de tareas son ideales para operaciones con poca carga, como tareas de red y de disco, además de tareas en las que el resultado deberá mostrarse al usuario en la actividad.

Durante un tiempo, desde la aparición de Kotlin, han coexistido diferentes métodos. Uno de ellos ha sido el uso de la librería **Anko**[1], creada y mantenida por **JetBrains**. Esta librería ha sido un complemento ideal, ya que facilitaba mucho la creación de tareas asíncronas, dejándolo en un mero trámite. Pero, a pesar de la facilidad que ofrecía *Anko*, fue *deprecated* en diciembre de 2019, aún así, puedes encontrar numerosos ejemplos que hacen uso de ella. *Anko* es reemplazado por la librería **Android KTX**[2] desarrollada por Google y forma parte de **Android JetPack**[3].

En el proyecto que se muestra a continuación, podrá verse como crear tareas asíncronas haciendo uso de las instrucciones básicas que ofrece Android. Se utilizará para ello la clase **AsyncTask**[4], pero cuidado, queda en desuso a partir de la API 30, se verá como solucionar esto más adelante.

Para este ejemplo, la actividad principal contendrá cuatro botones, dos para lanzar las tareas y dos para cancelarlas, así como dos barras de progreso para poder ver la evolución de las tareas lanzadas. A continuación, se creará una clase nueva llamada `MyAsyncTask`. Para poder interactuar con elementos de la UI, como el caso de la barra de progreso, se pasarán como parámetros los componentes que interesen, no será necesario pasar el contexto ya que se puede obtener a través de las vistas.

```
1  @Suppress("DEPRECATION")
2  class MyAsyncTask(progressBar: ProgressBar, btnStart: Button, btnCancel: Button) :
3      AsyncTask<Int, Int, Int>() {
4      // Se evita las pérdidas de memoria debido al contexto de las vistas.
5      private val progressBar: WeakReference<ProgressBar> = WeakReference(progressBar)
6      private val btnStart: WeakReference<Button> = WeakReference(btnStart)
7      private val btnCancel: WeakReference<Button> = WeakReference(btnCancel)
8
9      // Antes de iniciar la tarea, se utiliza para inicializar o configurar la tarea.
10     @Deprecated("Deprecated in Java")
11     override fun onPreExecute() {
12         Log.d(btnStart.get()!!.text.toString(), "${btnStart.get()!!.text}, iniciada!!")
13         progressBar.get()!!.progress = 0
14         btnStart.get()!!.isEnabled = false
15         btnCancel.get()!!.isEnabled = true
16     }
```

1 Anko (https://github.com/Kotlin/anko)
2 Android KTX (https://developer.android.com/kotlin/ktx)
3 Android JetPack (https://developer.android.com/jetpack)
4 AsyncTask (https://developer.android.com/reference/android/os/AsyncTask)

```
    // Este método realiza la tarea a ejecutar, su implementación es obligatoria.
    // vararg -> array con los parámetros indicados.
    @Deprecated("Deprecated in Java")
    override fun doInBackground(vararg p0: Int?): Int {
        if (p0.size == 2) {
            var contador = 0
            while (contador < p0[0]!!) {
                try {
                    contador++
                    Thread.sleep(p0[1]!!.toLong())
                } catch (e: Exception) {
                    Log.println(Log.WARN, "doInBackground", e.message.toString())
                }
                // Se comprueba si la tarea ha sido cancelada.
                if (!isCancelled)
                    publishProgress((((contador + 1) * 100 / p0[0]!!).toFloat()).toInt())
                else break
            }
            return 1
        } else return -1
    }

    // Método que se ejecutará al cancelarse la tarea.
    @Deprecated("Deprecated in Java")
    override fun onCancelled(result: Int?) {
        val ctxt = progressBar.get()!!.context

        progressBar.get()!!.progress = 0
        Log.d(btnStart.get()!!.text.toString(), "${btnStart.get()!!.text}, cancelada!!")
        Toast.makeText(
            ctxt, "${btnStart.get()!!.text}, cancelada!!", Toast.LENGTH_SHORT
        ).show()

        btnStart.get()!!.isEnabled = true
        btnCancel.get()!!.isEnabled = false
    }

    // Permite mostrar información al usuario, se ejecuta cuando se utiliza
    // el método publishProgress() desde el método doInBackground().
    @Deprecated("Deprecated in Java")
    override fun onProgressUpdate(vararg values: Int?) {
        progressBar.get()!!.progress = values[0]!!
    }

    // Se ejecuta al finalizar el método doInBackground() y se le pasa el
    // resultado obtenido.
    @Deprecated("Deprecated in Java")
    override fun onPostExecute(result: Int?) {
        if (result == 1) {
            val ctxt = progressBar.get()!!.context
```

```
            progressBar.get()!!.progress = 100
            Toast.makeText(ctxt, btnStart.get()!!.text, Toast.LENGTH_SHORT).show()
        }

        btnStart.get()!!.isEnabled = true
        btnCancel.get()!!.isEnabled = false
    }
}
```

Evidentemente, no es necesario conocer todos los métodos de la clase `AsyncTask`, una vez declarada la clase, basta con utilizar la opción *Override Methods* del menú *Code*, o el botón derecho sobre el código y seleccionar *Generate*, tras lo cual deberás seleccionar los métodos que desees implementar.

Si te fijas en `AsyncTask<Int, Int, Int>`, para este caso, los parámetros de tipo utilizados son de tipo entero, estos se podrán adaptar según las necesidades. Cada uno de ellos representa la siguiente información `<Params, Progress, Result>`.

Como ya se ha comentado, *AsyncTask* está *deprecated*, por lo se muestra con fines meramente educativos. Observa como se han recepcionado los parámetros, haciendo uso de la clase `WeakReference`, esta evita las pérdidas de memoria producidas por el contexto.

Observa ahora como quedaría el método `onCreate()` de la clase principal para poder crear las tareas y lanzarlas o cancelarlas.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    // Se deshabilitan los botones cancelar hasta que se activen las tareas.
    binding.btn1cancel.isEnabled = false
    binding.btn2cancel.isEnabled = false

    // Variables para crear las tareas asíncronas.
    lateinit var myAsyncTask1: MyAsyncTask
    lateinit var myAsyncTask2: MyAsyncTask
    binding.btn1.setOnClickListener {
      myAsyncTask1 = MyAsyncTask(binding.progressBar1, binding.btn1, binding.btn1cancel)
      // Se lanza la tarea.
      myAsyncTask1.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, 100, 20)
    }
    binding.btn1cancel.setOnClickListener {
      myAsyncTask1.cancel(true)
    }
    binding.btn2.setOnClickListener {
      myAsyncTask2 = MyAsyncTask(binding.progressBar2, binding.btn2, binding.btn2cancel)
        myAsyncTask2.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, 200, 15)
    }
```

```
    binding.btn2cancel.setOnClickListener {
        myAsyncTask2.cancel(true)
    }
}
```

Si ejecutas este código, verás como puedes lanzar las dos tareas y se producirá la ejecución en paralelo de ambas.

10.4. Corrutinas

Como se ha comentado anteriormente, *AsyncTask* quedó en desuso a partir de la API 30 y, junto con la aparición de *Android KTX*, desde la versión 1.3 de *Kotlin*, se añade lo que se conoce como *corrutinas*[56].

Las *corrutinas* son patrones de diseño que permiten simplificar el código que se quiere ejecutar de manera asíncrona. El objetivo principal de las *corrutinas* en Android es solucionar dos problemas:

- Evitar el bloque del proceso principal, impidiendo el parón de la aplicación, administrando más eficientemente las tareas largas.
- Aumentar la seguridad del sub-proceso principal, siendo más seguras las operaciones de disco o de red.

Años atrás, para poder utilizar *corrutinas* en un proyecto Android, debía añadirse la siguiente dependencia al `build.gradle (Module: app)`. Pero ahora ya están integradas, por lo que no es necesaria.

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.4'
```

Existen varias formas de crear *corrutinas*, este es un campo muy amplio, pero dada la naturaleza de este libro, no se puede profundizar mucho en ellas, pero sí tratar la que posiblemente sea la forma más utilizada y fácil de comprender de todas ellas, el *builder* ***launch***.

Para hacer uso de las *corrutinas* es necesario conocer que éstas se basan en las llamadas **funciones de suspensión**. Estas funciones, que únicamente pueden utilizarse dentro de una *corrutina*, pueden detener la ejecución de la *corrutina* y devolverle el control tras finalizar su tarea, convirtiendo las *corrutinas* en ese lugar seguro en el que no se bloqueará el hilo principal de la aplicación. Para crear una función de suspensión bastará con indicarlo haciendo uso de la palabra reservada `suspend`.

```
// Función de suspensión que detiene la ejecución de la corrutina hasta que finaliza.
suspend fun task(duracion: Long): Boolean {
    Log.d("SUSPEND FUN", "¡Simulando una tarea!")
    delay(duracion)
    return true
}
```

5 *Coroutines* (https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html)
6 Cómo mejorar el rendimiento de la app con las corrutinas de Kotlin (https://developer.android.com/kotlin/coroutines)

Las *corrutinas* se ejecutan bajo un contexto, generalmente el principal, este contexto define las reglas de ejecución. Estas reglas son un *Map*, por tanto funcionan como clave-valor, y lo que resulta más interesante de conocer es el *dispatcher*, que permite identificar el hilo en el que se ejecutará. Debes tener en cuenta que un hilo puede ejecutar múltiples *corrutinas*, esto significa que el sistema tratará de optimizar al máximo los hilos existentes, no creará nuevos hilos para ejecutar las *corrutinas*.

Volviendo a los *dispatchers*, estos serán diferentes tipos de contexto que especificarán el tipo de hilo que se encargará de ejecutar la tarea. A continuación se describen los cuatro tipos de *dispatchers* existentes.

- **Default**: se utiliza si no se especifica lo contrario, es para tareas que hacen un uso intensivo de la CPU, cálculos de la aplicación por ejemplo.
- **IO**: corresponde a las operaciones de entrada/salida, son tareas que bloquean el hilo hasta recibir una respuesta, como un acceso a base de datos, lectura de un fichero, peticiones a un servidor, etc.
- **Unconfined**: este se utilizará cuando el hilo en el que se ejecute no es importante.
- **Main**: este *dispatcher* utiliza librerías relacionadas con la interfaz de usuario, hace uso del hilo de UI, por ejemplo, para actualizar una barra de progreso.

Como ya se ha comentado, se utilizará el *builder* ***launch*** para la creación de *corrutinas*, éste necesita de un *scope* para funcionar, se utilizará ***CoroutineScope*** que trabaja sobre *GlobalScope*, que se utiliza para lanzar *corrutinas* de nivel superior y trabajan durante toda la vida útil de la aplicación y no se cancelan prematuramente.

Debes saber que este *builder* devolverá un objeto `Job` que contiene dos funciones que debes conocer.

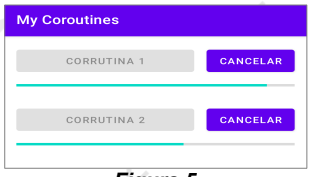

Figura 5

- **join()**: bloqueará la *corrutina* asociada hasta que todos sus hijos finalicen. Debe utilizarse dentro de una *corrutina*, ya que se trata de una función de suspensión.
- **cancel()**: cancela la ejecución de los hijos de una *corrutina*.

Fíjate en el código que viene a continuación para comprender la creación y funcionamiento de las *corrutinas*.

```
1  private fun makeTask(
2      duracion: Int, btnStart: Button, btnCancel: Button, progressBar: ProgressBar
3  ): Job = CoroutineScope(Dispatchers.Main).launch {
4      // Preparación de la corrutina.
5      btnStart.isEnabled = false
6      btnCancel.isEnabled = true
7      progressBar.progress = 0
8
9      withContext(Dispatchers.IO) { // Tarea principal.
10         var contador = 0
11         while (contador < duracion) {
```

```
            if (task((duracion * 50).toLong())) {
                contador++
                progressBar.progress = (contador * 100) / duracion
            }
        }
    }

    // Finaliza la corrutina.
    btnStart.isEnabled = true
    btnCancel.isEnabled = false
    progressBar.progress = 0
    Toast.makeText(
        this@MainActivity, "${btnStart.text} finalizada!!", Toast.LENGTH_SHORT
    ).show()
}
```

Observa ahora la llamada que deberá hacerse para crear la *corrutina* desde un botón en la actividad principal de la aplicación, recogiendo el *Job* que se crea. También puedes ver como se cancelaría la tarea.

```
// Las corrutinas devuelven un objeto Job al crearse.
var task1: Job? = null

binding.btn1.setOnClickListener {
    task1 = makeTask(10, binding.btn1, binding.btn1cancel, binding.progressBar1)
}

binding.btn1cancel.setOnClickListener {
    // LET ejecutará el contenido si task1 existe y/o no es nulo.
    task1.let {
        task1?.cancel().apply {
            Toast.makeText(
                this@MainActivity,
                "${getString(R.string.txt_btn_1)} cancelada!!",
                Toast.LENGTH_SHORT
            ).show()
        }

        binding.progressBar1.progress = 0
        binding.btn1cancel.isEnabled = false
        binding.btn1.isEnabled = true
    }
}
```

Como verás más adelante, hay otras formas de lanzar *coroutines* sin utilizar *CoroutineScope*, como el *launch* del *ViewModel*, por ejemplo.

10.5. Notificaciones

Las notificaciones son un elemento de Android permite mostrar información de una manera breve al usuario en la barra de estado del dispositivo. A continuación, se verá como crear notificaciones básicas, utilizando para ello características que pertenecen a la API 4 y superiores, concretamente, se hará hincapié en el cambio sufrido a partir de la API 26 con la aparición de los canales de notificación.

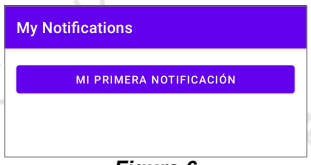

Figura 6

Se empezará creando un proyecto nuevo, que tendrá un único botón para comenzar, el cual, cada vez que se pulse creará una notificación que aparecerá en la barra de estado. Evidentemente, este no es el cometido de las notificaciones, ya que se utilizan más para notificar tareas acabas, mensajes recibidos, etc. Pero para poder ver como se crean y su funcionamiento, servirá.

Ya en el código, se añadirá la siguiente propiedad a la clase principal, esta define el nombre que tendrá el canal de notificaciones, suele ser el nombre de la aplicación.

```
1  private val CHANNELID1 = "es.javiercarrasco.mynotifications"
```

En el método `onCreate()` añade el siguiente código, éste utiliza el *builder* de la clase `NotificationCompat` para crear la notificación, al que se le pasará el contexto y la variable `CHANNELID1` creada.

```
1  // Se crea una notificación utilizando el builder de NotificationCompat.
2  private val builder = NotificationCompat.Builder(this, CHANNELID1).apply {
3      setSmallIcon(R.drawable.notificacion)
4      setContentTitle("Mi primera notificación")
5      setContentText("Esta será la primera notificación creada.")
6      priority = NotificationCompat.PRIORITY_DEFAULT
7  }
```

Volviendo al objeto `builder`, la creación es bastante intuitiva, se asigna un icono a la notificación, un título y un contenido o descripción de la notificación. Por último, la prioridad que tendrá, *PRIORITY_DEFAULT*[7].

El **canal**[8] es algo que debe utilizarse siempre que el objetivo de la aplicación sea de la API 26 (Android 8.0) en adelante, a partir de esta versión, el sistema permite que el usuario tenga el control de las notificaciones de la aplicación. Básicamente permiten distinguir entre diferentes tipos de notificaciones dentro de una misma aplicación. De esta manera se dejará a elección del usuario que notificaciones desea recibir, aunque dependerá principalmente de la activación o no por parte del desarrollador.

Para los siguientes casos se utilizará un emulador con **API 28** como mínimo para poder ver los canales en la gestión de notificaciones. Deberá registrarse al menos un canal de notificaciones para la aplicación, de lo contrario, no sería posible mostrar notificaciones, y cada vez que se intentase mostrar una, en el *log* podría verse el aviso de canal no registrado para la aplicación.

7 Prioridades (https://developer.android.com/reference/androidx/core/app/NotificationCompat.html)
8 Crear canales en Android (https://developer.android.com/training/notify-user/channels#CreateChannel)

A continuación, se añadirá el método `createNotificationChannel()` que se encargará de crear los canales.

```
private fun createNotificationChannel(
    channel: String, name: Int, desc: Int, importance: Int
) {
    // Se crea el canal de notificación únicamente para API 26+.
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val mName = getString(name)
        val descriptionText = getString(desc)
        val mChannel = NotificationChannel(channel, mName, importance)
        mChannel.description = descriptionText

        // Se registra el canal en el sistema.
        val notificationManager: NotificationManager =
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(mChannel)
    }
}
```

Justo antes de crear la notificación, se deberá registrar el canal en el sistema operativo. Esto puede hacerse nada más iniciar la aplicación. Fíjate ahora como se lanzaría la notificación creada, en este caso, asociada a la pulsación sobre el botón creado.

```
binding.button.setOnClickListener {
    // Se registra el canal de notificación en el sistema.
    createNotificationChannel(
        CHANNELID1,
        R.string.txt_channel1_name,
        R.string.txt_channel1_desc,
        NotificationManager.IMPORTANCE_DEFAULT
    )
    with(NotificationManagerCompat.from(this)) {
        notify(notificationId1, builder.build())
    }
}
```

La variable *notificationId1* contendrá el número que identifique a la notificación lanzada, este deberá ser diferente si las notificaciones son distintas, por ejemplo, un código para notificar una descarga, otra para recibir un mensaje, etc.

Para este ejemplo, el código utilizado es un número al azar, como el que se muestra a continuación.

```
private val notificationId1 = 1
```

Pero recuerda que, si tu aplicación lanza más de una notificación diferente, o la misma varias veces, estas deberán tener distintos identificadores para poder tratarlas según la acción que se haga sobre ellas.

El resultado de este ejemplo lo puedes ver en la figura, si reproduces el código, verás que, por mucho que pulses el botón, únicamente aparecerá una notificación, esto es debido al identificador. Si tu API objetivo es inferior a la 26, puedes ahorrarte la comprobación de versión y omitir la creación del canal, pero no está de más.

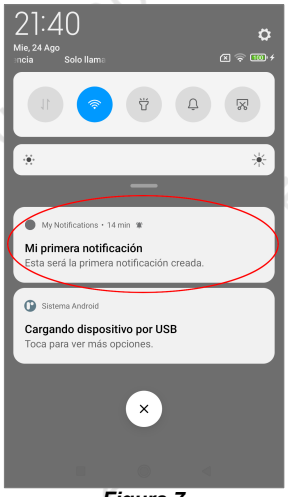

Figura 7

Algo que deberás tener presente es lo siguiente, a partir de Android 13 (nivel de API 33) y posteriores se necesitará un permiso de tiempo de ejecución para enviar notificaciones desde primer plano, el cual será necesario que conceda el usuario si quieres enviar notificaciones desde tu aplicación.

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

Habrá que comprobar en que versión te encuentras para tomas el camino necesario, por ejemplo.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
    // Se comprueba el permiso para lanzar la notificación.
} else {
    with(NotificationManagerCompat.from(this)) {
        notify(notificationId1, builder.build())
    }
}
```

Ya puedes lanzar una notificación, pero si la pruebas, verás que al hacer clic sobre ella no pasa nada, eso es porque hay que decirle a la notificación qué hacer, que será indicarle que *intent* debe abrir al pulsar sobre ella. Crea una nueva actividad vacía llamada `RequestNotification`. El *layout* únicamente contendrá un *TextView* con el texto "*Has recibido una notificación*".

Ahora, debes decirle a la notificación, que al crearse, tendrá un *intent* pendiente, esto hará que al pulsar sobre la notificación se abra la *activity* que se le indique. Estas líneas van justo antes de invocar el método *build* del objeto `builder`.

```
// Se crea el intent que deberá abrirse al pulsarse la notificación.
val intent = Intent(this, RequestNotification::class.java).apply {
    flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
}
val pendingIntent: PendingIntent =
    PendingIntent.getActivity(this, notificationId1, intent, PendingIntent.FLAG_IMMUTABLE)
```

El tercer parámetro de *PendingIntent.getActivity* es *requestCode*, se utiliza para identificar el código de respuesta, generalmente se utiliza el mismo identificador que el utilizado al crear la notificación. El cuarto parámetro indica que el *PendingIntent* será inmutable, ignorando cualquier otro parámetro, existen otro *flags* que permiten modificar el comportamiento de los *PendingIntents*[9].

Seguidamente, se establecen en el objeto builder las siguientes dos propiedades.

9 *PendingIntents* (https://developer.android.com/reference/android/app/PendingIntent)

```
builder.setContentIntent(pendingIntent)
builder.setAutoCancel(true)
```

Se utiliza la propiedad `setContentIntent()` para especificar el *intent* que debe abrirse y, la propiedad `setAutoCancel()` a *true* para que se cierre la notificación una vez pulsada.

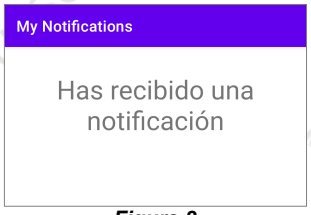

Figura 8

Debes saber que este *intent* está fuera del ciclo de vida de la aplicación, por lo que si utilizas la propiedad *parent* en el *manifest* no funcionará. Tras pulsar sobre la notificación, verás como se abre la nueva actividad.

Las siguientes imágenes muestran el canal de notificaciones de la aplicación a través de la información de la aplicación del sistema, que sería lo que se tendría por defecto si no se crean más canales. Estas imágenes pueden variar según la versión del sistema y su capa de personalización.

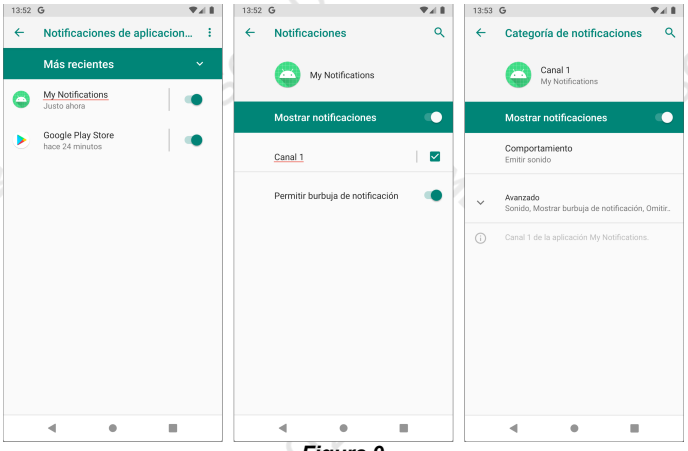

Figura 9

Puedes adaptar el ejemplo de las *Coroutines* visto antes de forma que cuando finalice la tarea lance una notificación en lugar de un *Toast*. Puedes comprobar que quitando la aplicación del primer plano, sin cerrarla, verás como las notificaciones avisan de la finalización de la tarea.

10.6. Un ejemplo útil, descarga de imágenes

En este punto se verá un ejemplo con cierta utilidad, se creará una aplicación que permita descargar una serie de imágenes desde Internet y mostrarlas en pantalla, además, se realizará una acción conocida como *"inflado"*, y se añadirá una comprobación para saber si hay red para poder hacer la descarga. Se creará un nuevo proyecto llamado *"My Image Downloader"* y se indicará en el fichero *AndroidManifest.xml* el permiso de uso de Internet.

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

Como también se hará uso de la librería *Glide* para cargar las imágenes, deberás añadir al *Gradle (Module:app)* la siguiente dependencia.

```
implementation 'com.github.bumptech.glide:glide:4.14.2'
```

La actividad principal de la aplicación contendrá como contenedor raíz un *ScrollView*, dentro del cual habrá un *LinearLayout* vertical que contendrá; una botón para iniciar la descarga, una *ProgressBar* horizontal que marcará el progreso de la descarga, un *TextView* para mostrar información y por último, otra *ProgressBar* circular que se utilizará para indicar que está trabajando. El código XML para formar la UI puede ser parecido al siguiente.

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:id="@+id/linearLayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="8dp">

        <com.google.android.material.button.MaterialButton
            android:id="@+id/button"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/btn_imageDownloader" />

        <ProgressBar
            android:id="@+id/progressBar"
            style="?android:attr/progressBarStyleHorizontal"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

        <TextView
            android:id="@+id/textInfo"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:gravity="center"
            tools:text="información" />

        <ProgressBar
            android:id="@+id/progressBarURL"
            style="?android:attr/progressBarStyle"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:visibility="gone" />
    </LinearLayout>
</ScrollView>
```

Fíjate que únicamente se utilizará un botón que cambiará de estado al pulsarse, una barra de progreso para mostrar el estado de la tarea, que en este casó será la descarga de imágenes, y un *TextView* que mostrará información sobre la tarea.

Observa que no hay ningún *ImageView* en la actividad, esto es porque se *"inflarán"* mediante código dentro del *LinearLayout*.

En primer lugar se creará una *mutableListOf* en la que se almacenarán las direcciones de las imágenes que se quieren descargar obtenidas desde un fichero RAW (consulta el Anexo I).

```
// Contendrá las URLs de las imágenes a descargar.
private val urlImages: MutableList<String> = mutableListOf()
```

El contenido del fichero `R.raw.urlimages`, que es un fichero de texto llamado `urlimages.txt` almacenado en la carpeta de recurso RAW contendrá las siguientes líneas.

```
https://live.staticflickr.com/7801/47562604261_4ff8522918_k.jpg
https://live.staticflickr.com/8345/28338120516_40e0c40f65_k.jpg
https://live.staticflickr.com/7449/28346482855_3d1b745518_k.jpg
https://live.staticflickr.com/3856/14648036959_72973ac0d3_k.jpg
https://live.staticflickr.com/5587/14648212427_b3bf236e4c_k.jpg
```

La aplicación leerá el fichero RAW al iniciarse, poblando así la lista de las URLs de las imágenes, una vez leído el fichero, habilitará el botón para iniciar la descarga de imágenes. Para la lectura se creará una *coroutine* formada por los dos siguientes métodos.

```
// Coroutine encargada de lanzar la lectura del fichero RAW.
private fun readURLs() = CoroutineScope(Dispatchers.Main).launch {
    // Preparación
    binding.progressBarURL.isVisible = true

    withContext(Dispatchers.IO) {
        readFile()
        // Retiene la ejecución para ver la UI.
        delay(2000)
    }

    // Fin de la lectura del fichero
    Log.i("FILE", urlImages.size.toString())
    binding.button.isEnabled = urlImages.size > 0
    binding.progressBarURL.isVisible = false
}

// Lee el fichero RAW de las URLs para almacenarlas en la lista urlImages.
private fun readFile() {
    val entrada = InputStreamReader(resources.openRawResource(R.raw.urlimages))
    lateinit var br: BufferedReader

    try {
        br = BufferedReader(entrada)
```

```
        var linea = br.readLine()

        while (!linea.isNullOrEmpty()) {
            urlImages.add(linea)
            Log.i("FILE", linea)
            linea = br.readLine()
        }
    } catch (e: IOException) {
        Log.e("ERROR IO", e.message.toString())
    } finally {
        br.close()
        entrada.close()
    }
}
```

El método `onCreate()` únicamente contendrá la llamada al método `readURLs()`, de esta forma se poblará la lista durante la creación de la UI.

El siguiente paso será crear la *coroutine* encargada de la descarga de las imágenes, fíjate como se cambia de contexto, utilizando el *Dispatcher* a **IO** para realizar la operación de descarga, las imágenes se almacenarán en un *ArrayList* para su posterior tratamiento.

```
private fun downloadImages() = CoroutineScope(Dispatchers.Main).launch {
    val images = ArrayList<Bitmap>()

    binding.button.text = getString(android.R.string.cancel)
    binding.textInfo.text = getString(R.string.txt_descargando)
    binding.progressBar.progress = 0

    urlImages.forEach {
        Log.d("URLs", it)

        // Se descargan las imágenes y se almacenan en un ArrayList<Bitmatp>().
        withContext(Dispatchers.IO) {
            try {
                val inputStream = URL(it).openStream()
                images.add(BitmapFactory.decodeStream(inputStream))
                inputStream.close()
            } catch (e: Exception) {
                Log.e("DOWNLOAD", e.message.toString())
            }
        }

        binding.progressBar.progress = (images.size * 100) / urlImages.size
    }

    // Fin de la tarea. Se añaden las imágenes a la vista una vez descargadas.
    images.forEach {
        addImage(it)
    }
```

```
    binding.button.text = getString(R.string.btn_imageDownloader)
    binding.textInfo.text = getString(R.string.txt_descargaCompleta, images.size)
}
```

A continuación, se detalla el método `addImage()` encargado de realizar el *"inflado"* de los *ImageView*. Recuerda que se ha añadido la dependencia necesaria para utilizar *Glide* al inicio.

```
private fun addImage(image: Bitmap) {
    val img = ImageView(this)

    // Se carga la imagen en el ImageView mediante Glide y se ajusta el tamaño.
    Glide.with(this)
        .load(image)
        .override(binding.linearLayout.width - 100)
        .into(img)
    img.setPadding(0, 0, 0, 10)

    // Se infla el LinearLayout con una imagen nueva.
    binding.linearLayout.addView(img)
}
```

En este caso se utilizará el método `onStart()` para añadir la funcionalidad, esto se hace para dar tiempo a la aplicación a terminar de cargar las URLs. El código quedará como se muestra a continuación, se encargará de cambiar el estado del botón, limpiar el *LinearLayout* y crear y cancelar la tarea.

```
override fun onStart() {
    super.onStart()
    var tarea: Job? = null

    binding.button.setOnClickListener {
        if (binding.button.text == getString(R.string.btn_imageDownloader)) {
            // Se comprueba la existencia de ImageViews, si existen se eliminan.
            if (binding.linearLayout.childCount > 3) {
                binding.linearLayout.removeViews(
                    3,
                    (binding.linearLayout.childCount) - 3
                )
            }
            tarea = downloadImages()
        } else {
            tarea?.let {
                tarea?.cancel()
                binding.button.text = getString(R.string.btn_imageDownloader)
                binding.textInfo.text = getString(R.string.txt_descargaCancelada)
            }
        }
    }
}
```

10.6.1. Comprobar el estado de la conexión

Por último, se creará un nuevo método llamado `checkConnection()`, este permitirá realizar una comprobación del estado de la conexión y saber si se puede o no realizar la operación de descarga. Lo primero que debe hacerse para poder realizar esta comprobación es añadir el permiso necesario en el *manifest*.

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

El método hará uso del objeto *ConnectivityManager* para obtener la información necesaria para la comprobación, la idea es devolver *true* cuando exista cualquiera de los tres tipos de conexión evaluados.

```
private fun checkConnection(): Boolean {
    var estado = false

    // Se comprueba el estado de la conexión.
    val cm = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    val networkInfo = cm.activeNetwork
    val actNetwork: NetworkCapabilities? = cm.getNetworkCapabilities(networkInfo)

    if (actNetwork != null)
        estado = when {
            actNetwork.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) -> true
            actNetwork.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET) -> true
            actNetwork.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) -> true
            else -> false
        }
    return estado
}
```

En la siguiente figura puedes observar el resultado del código que se ha descrito sobre estas líneas. La segunda captura muestra el *SnackBar* que indica que el dispositivo no tiene conexión para realizar la operación de descarga.

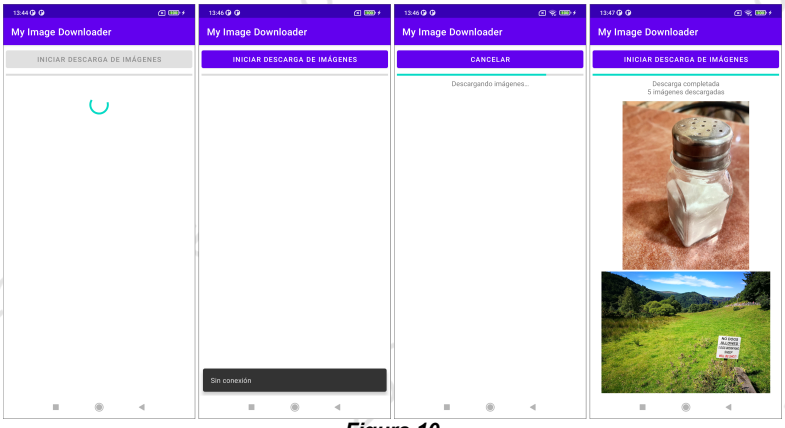

Figura 10

10.7. Uso del vibrador

Cuando se recibe una notificación, finaliza una tarea, etc, por lo general, se suele hacer vibrar el dispositivo. Para poder hacer uso del vibrador, al tratarse de un recurso físico del dispositivo, debe indicarse el permiso en el *manifest* del proyecto.

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

Se creará un método con el código necesario para hacer vibrar el dispositivo. Evidentemente, esto no ocurrirá si se prueba en el emulador, para notarlo deberás utilizar un dispositivo físico. Esta vibración no vibrará al entrar la notificación, ya que eso dependerá del sistema operativo, si no que lo hará en el momento en el que se pulse un botón, acabe una tarea, según sea necesario.

```
private fun vibrate() {
    // Se instancia el objeto Vibrator según la API utilizada
    val vibrator: Vibrator
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        val vibratorManager = getSystemService(VIBRATOR_MANAGER_SERVICE) as VibratorManager
        vibrator = vibratorManager.defaultVibrator
    } else {
        vibrator = getSystemService(VIBRATOR_SERVICE) as Vibrator // Deprecated API 31
    }

    // Se comprueba la existencia de vibrador.
    if (!vibrator.hasVibrator())
        Toast.makeText(this, "No tienes vibrador!!", Toast.LENGTH_SHORT).show()
    else {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val vibrationEffect = VibrationEffect.createOneShot(
                1000,
                VibrationEffect.DEFAULT_AMPLITUDE
            )
            vibrator.vibrate(vibrationEffect)
        } else {
            vibrator.vibrate(1000) // Deprecated API 26
            Toast.makeText(this, "Deprecated version", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Fíjate que no es necesario pedir permiso al usuario para poder utilizar la vibración del dispositivo, pero si debe controlarse la versión para utilizar el método `vibrate()`, ya que desde la API 26 está *deprecated*, así como al crear la variable ***vibrator***, el *string* VIBRATOR_SERVICE aparece marcado como *deprecated*, esto se produce a partir de la API 31, pasándose a utilizar VIBRATOR_MANAGER_SERVICE y la clase *VibratorManager*.

Ya solo faltaría añadir la llamada al método `vibrate()`, por ejemplo, en la aplicación de descarga de imágenes, se puede hacer vibrar el dispositivo cuando terminen de descargarse todas las imágenes.

Si en algún momento necesitas cancelar la vibración, basta con utilizar el método `cancel()` del objeto `Vibrator` creado, por ejemplo, al salir de la aplicación.

```
override fun onDestroy() {
    super.onDestroy()
    vibrador.cancel()
}
```

10.8. Reproducir un sonido

Para reproducir un sonido propio de la aplicación, este deberá encontrarse en la carpeta RAW del proyecto, para ello, crea el recurso y añade un archivo de audio.

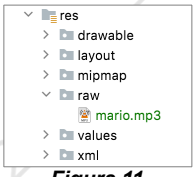

Figura 11

Para hacer sonar el audio habrá que hacer uso de la clase `MediaPlayer`, al reproducir un audio local, puedes evitarte la preparación, sincronización, etc, algo que debe hacerse si la fuente de audio estuviese en otra localización.

Las siguientes líneas reproducirán el fichero de audio `mario.mp3` localizado en el recurso RAW del proyecto.

```
val mediaPlayer = MediaPlayer.create(applicationContext, R.raw.mario)
mediaPlayer.start()
```

Esta sería la versión más básica para reproducir un sonido, no necesita preparación ni sincronización. Además del método `start()`, también dispones de métodos como `pause()` y `stop()` para controlar la reproducción.