

Tema 6. Dialogs, tareas asíncronas y notificaciones

- 6.1 Dialogs
- 6.2 Usando AlertDialog
- 6.3 Utilizando DialogFragment
 - 6.3.1 Cuadros de diálogo personalizados
- 6.4 Time y DatePicker
- 6.5 Programación Multihilo en Android
 - 6.5.1 Tareas asíncronas - AsyncTask (Obsoleto)
 - 6.5.2 Hilos con Thread y Handler
 - 6.5.3 Corrutinas (Coroutines) → El más recomendado
 - 6.5.4 Progress Bar
 - 6.5.5 SplashScreen
 - 6.5.6 Descarga de imágenes → Librería Glide
- 6.6 Notificaciones
- 6.7 Vibración
- 6.8 Reproducción de sonidos

6.1 Dialogs

Un Dialog es una ventana de tamaño reducido con la que se indicará al usuario que debe tomar una decisión o introducir algún tipo de información.

También pueden usarse para una confirmación rápida, solicitar una elección entre varias, etc.

La forma más sencilla de usarlos es utilizando la subclase **AlertDialog** de la clase base **Dialog**. Aquí tienes la documentación oficial: <https://developer.android.com/guide/topics/ui/dialogs?hl=es-419>

También puedes encontrar cómo usarlos en la web **material.io**: <https://m2.material.io/components/dialogs#alert-dialog>

Se tienen 3 áreas en un **AlertDialog**:

1 - Título

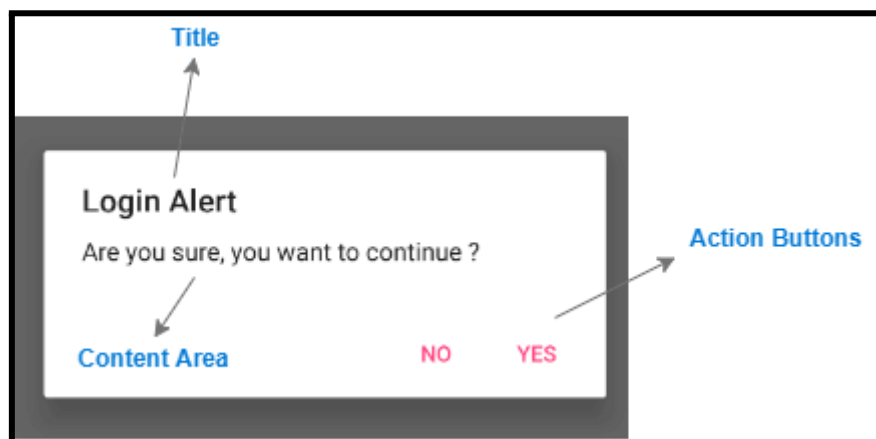
2 - Área de contenido

3 - Área de botones

3.1 - **Positivo**. Puedes usar este botón para aceptar y continuar con la acción (la acción "Aceptar").

3.2 - **Negativo**. Este botón se utiliza para cancelar la acción.

3.3 - **Neutral**. Se utiliza para no continuar la acción, pero sin cancelar. Por ejemplo, la acción podría ser "Recordarme más tarde".



6.2 Usando AlertDialog

Es posible utilizar AlertDialog directamente sin necesidad de crear una clase.

```

//AlertDialog
val builder = AlertDialog.Builder(context: this)
builder.setTitle(R.string.titulo)
    .setMessage(R.string.alert)
    .setPositiveButton(R.string.start,
        DialogInterface.OnClickListener {
            //Se lanza el activity si se pulsa Sí
            dialog, id -> startActivity(intent)
        })
    .setNegativeButton(R.string.cancel,
        DialogInterface.OnClickListener { dialog, id -> Toast.makeText(context: this, text: "Has cancelado el paso a SegundoActivity",
            Toast.LENGTH_LONG
        ).show()
        })

// Create the AlertDialog object and return it
builder.create()
builder.show()
}

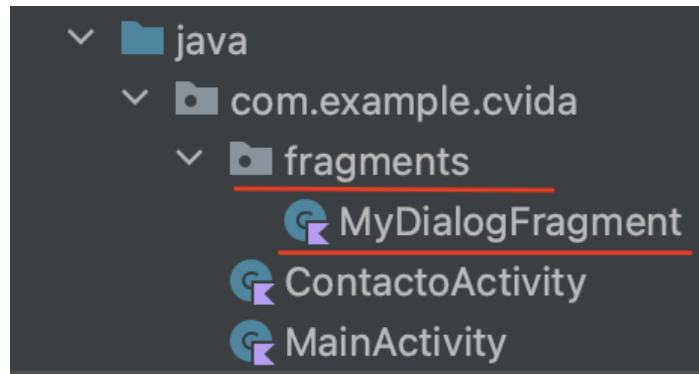
```



6.3 Utilizando DialogFragment

En este caso, vamos a ver cómo generar un *AlertDialog*, a partir de la redefinición de la clase *DialogFragment*.

Para ellos nos vamos al menú File → New → Kotlin File / Class. Se ha creado un nuevo paquete donde almacenar todos los *Fragments*.



El código a desarrollar dentro de la clase *MyDialogFragment* es el siguiente:

```
class MyDialogFragment: DialogFragment()
{
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog
    {
        return activity?.let { it: FragmentActivity

            val builder = AlertDialog.Builder(it)

            builder.setMessage(R.string.my_first_dialog)

            builder.setPositiveButton(android.R.string.ok) {
                dialog, which ->
                //Acciones si se pulsa SÍ
                Log.d( tag: "DEBUG", msg: "SÍ")

                val intent: Intent = Intent(context, ContactoActivity::class.java).apply { this: Intent
                    putExtra( name: "PHONENUMBER", value: "666111222")
                }
                startActivity(intent)
            }

            builder.setNegativeButton(android.R.string.cancel) {
                dialog, which ->
                //Acciones si se pulsa NO
                Log.d( tag: "DEBUG", msg: "NO")
                this.dismiss()
            }

            builder.create() ^let
        } ?: throw IllegalStateException("El Activity no puede ser nulo")
    }
}
```

Notas importantes:

- **activity?.let {...}**: Utiliza el operador **?.let** para realizar acciones solo si activity no es nulo. El código dentro de let se ejecuta solo si la variable activity no es nula.

- **?: throw IllegalStateException(...)**: Esta expresión maneja el caso en que *activity* sea nulo. Si *activity* es nulo, lanza una excepción *IllegalStateException*.
- **Botón Positivo (setPositiveButton)**: Configura el botón "Aceptar" del cuadro de diálogo. Cuando se pulsa, se ejecuta el bloque de código que incluye abrir la actividad *ContactoActivity* con un número de teléfono como dato extra.
- **Botón Negativo (setNegativeButton)**: Configura el botón "Cancelar" del cuadro de diálogo. Cuando se pulsa, se ejecuta el bloque de código que cierra el *DialogFragment* usando *this.dismiss()*.
- **it** se refiere al objeto *activity* que se verifica si es nulo antes de entrar en el bloque *let*. El operador **?.let** se usa para ejecutar el bloque de código dentro de *let* solo si *activity* no es nulo. La instancia de *AlertDialog.Builder* se crea utilizando *it*, que en este caso es la referencia al objeto *activity*.

```
kotlin
{ dialog, which ->
    // ...
}
```

- **`dialog`**: Es una referencia al propio cuadro de diálogo (**`AlertDialog`**) que se está mostrando.
- **`which`**: Es un código que identifica el botón que fue presionado. En el contexto de **`setPositiveButton`** y **`setNegativeButton`**, **`which`** generalmente toma los valores **`DialogInterface.BUTTON_POSITIVE`** o **`DialogInterface.BUTTON_NEGATIVE`**, respectivamente.

Una vez definido el *DialogFragment*, debemos ir al Activity desde el cuál queremos lanzarlo.

```
btn.setOnClickListener()
{
    it ->

    val myDialogFragment = MyDialogFragment()
    myDialogFragment.show(supportFragmentManager, tag: "Mi primer DialogFragment")
}
```

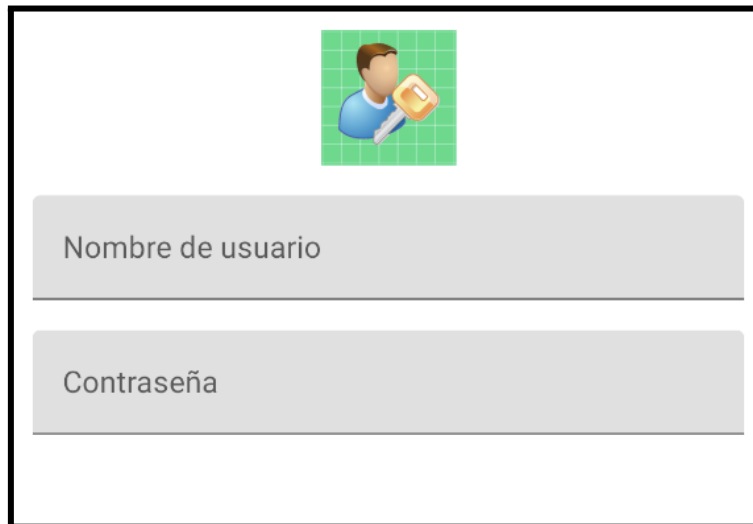
En el método *show()*, el segundo argumento *"Mi primer DialogFragment"*, es una etiqueta única que el sistema podrá guardar.

6.3.1 Cuadros de diálogo personalizados

En ocasiones, puede resultar útil mostrar un cuadro de diálogo con un diseño personalizado, que se adapte a unas necesidades concretas.

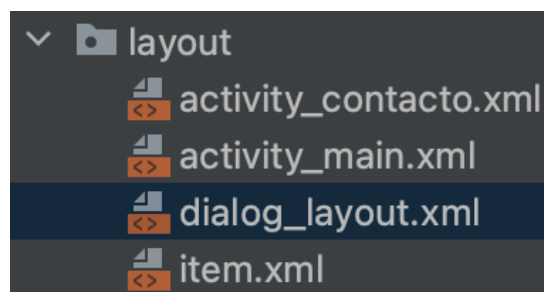
Para ello, se utilizará el método `setView()` para asignar un layout personalizado.

En primer lugar, se debe crear un diseño personalizado, para lo cual, debemos utilizar un fichero de tipo layout. Por ejemplo, podemos usar el siguiente diseño.



El diagrama muestra un cuadro de diálogo con un fondo blanco y una barra superior gris. En el centro superior hay un icono de un usuario con una llave. Debajo del icono hay dos campos de texto rectangulares con bordes grises. El primer campo contiene el texto "Nombre de usuario" y el segundo campo contiene el texto "Contraseña".

Definido en el fichero *dialog_layout.xml*, dentro de la carpeta de recursos layout.



Finalmente, solo necesitamos añadir esta línea, en el fichero donde definimos la clase *MyDialogFragment*.

```
//Cambiamos el formato del AlertDialog  
builder.setView(R.layout.dialog_layout)
```

6.4 Time y DatePicker

Un *Time Picker* permite seleccionar la hora de forma muy sencilla. Un *Date Picker*, funciona exactamente igual que Time Picker, pero en este caso, se usa para seleccionar una fecha.

Aquí tienes un ejemplo de uso de cada uno de ellos.

```
binding.btn1.setOnClickListener { it: View!>

    val cal = Calendar.getInstance()

    val timeSetListener = TimePickerDialog.OnTimeSetListener {
        _, hour, minute -> cal.set(Calendar.HOUR_OF_DAY, hour)
        cal.set(Calendar.MINUTE, minute)
        binding.tvTime.text = SimpleDateFormat( pattern: "HH:mm").format(cal.time)
    }

    TimePickerDialog( context: this@ContactoActivity,
        timeSetListener, cal.get(Calendar.HOUR_OF_DAY), cal.get(Calendar.MINUTE), is24HourView: true).show()
}

binding.btn2.setOnClickListener { it: View!>

    val cal = Calendar.getInstance()

    val dateSetListener = DatePickerDialog.OnDateSetListener {
        _, i, i2, i3 -> cal.set(Calendar.YEAR, i)
        cal.set(Calendar.MONTH, i2)
        cal.set(Calendar.DAY_OF_MONTH, i3)
        binding.tvDate.text = "${cal.get(Calendar.DAY_OF_MONTH)}" +
            "/" + "${cal.get(Calendar.MONTH)+1}" +
            "/" + "${cal.get(Calendar.YEAR)}"
    }

    DatePickerDialog( context: this@ContactoActivity,
        dateSetListener, cal.get(Calendar.YEAR), cal.get(Calendar.MONTH), cal.get(Calendar.DAY_OF_MONTH)).show()
}
```

- *TimePickerDialog.OnTimeSetListener* es una interfaz proporcionada por Android para escuchar eventos de configuración de tiempo en un *TimePickerDialog*.
- *{ _, hour, minute -> ... }* es la sintaxis de la expresión lambda. En este caso, la lambda toma tres parámetros. El guión bajo '_' se usa para indicar que no se va a utilizar el primer parámetro, que en este caso es la referencia al *TimePicker*.
- *cal.set(Calendar.HOUR_OF_DAY, hour)* y *cal.set(Calendar.MINUTE, minute)* son líneas de código que actualizan un objeto *Calendar* llamado cal con la hora y el minuto seleccionados por el usuario. El *Calendar.HOUR_OF_DAY* representa la hora del día en formato de 24 horas.

6.5 Programación Multihilo en Android

(Fuente: <https://www.develou.com/android-async-task-hilos/>)

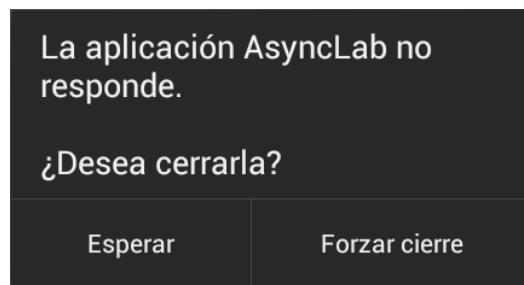
La programación multihilo en Android es esencial para realizar tareas concurrentes y mantener la interfaz de usuario (UI) receptiva.

Cuando se construye una aplicación Android, todos los componentes y tareas son introducidos en el hilo principal o *hilo de UI (UI Thread)*. Hasta el momento hemos trabajado de esta forma, ya que las operaciones que hemos realizado toman poco tiempo y no muestran problemas significativos de rendimiento visual en nuestros ejemplos.

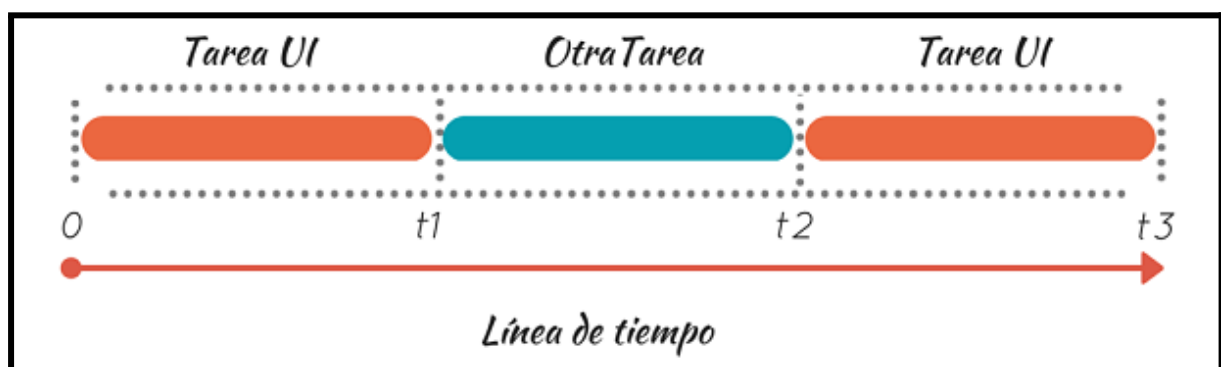
Pero en tus proyectos reales no puedes pretender que todas las acciones que lleva a cabo tu aplicación sean simples y concisas.

En ocasiones hay instrucciones que toman unos segundos en terminarse, esta es una de las mayores causas de la terminación abrupta de las aplicaciones. Android está diseñado para ofrecerle al usuario la opción de terminar aquellas aplicaciones que demoran más de 5 segundos en responder a los eventos del usuario.

Cuando esto pasa, podemos ver el famoso **Diálogo ANR** (Application not respond).

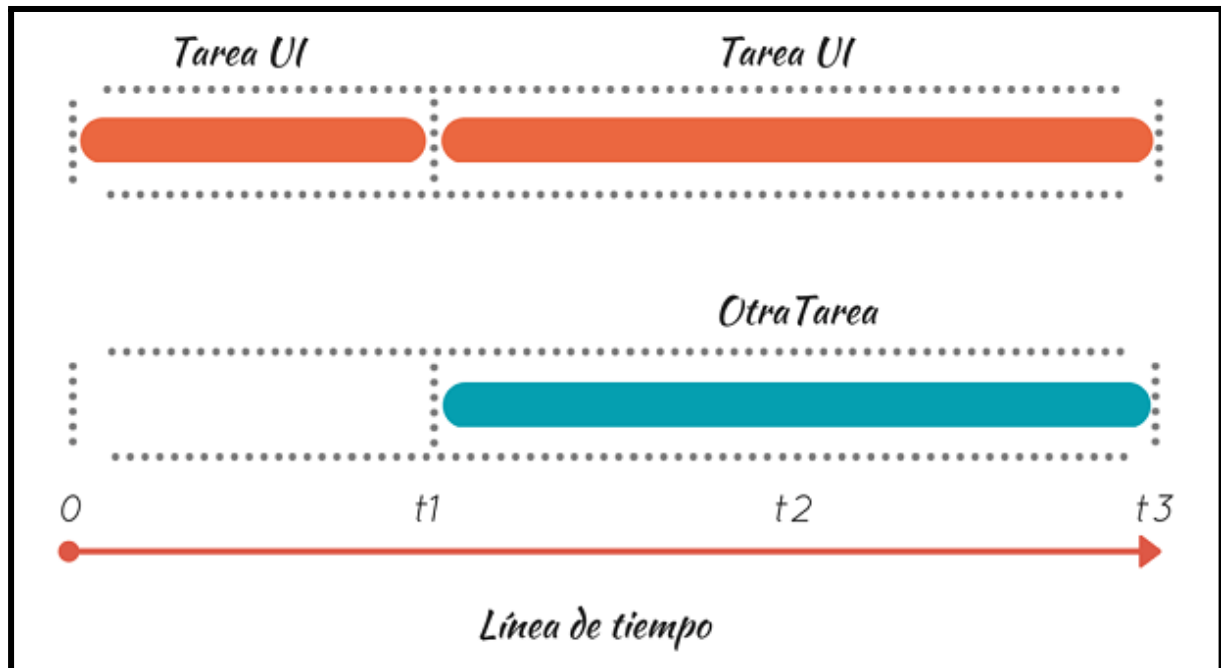


La ejecución usando únicamente *el hilo principal (UI Thread)* de diferentes tareas se realiza como en la imagen siguiente:



Si una tarea toma algunos segundos, se arruinaría la capacidad de respuesta, ya que las tareas están en serie, es decir, hasta que una no acabe la otra no puede iniciar.

El camino correcto es renderizar la interfaz de la aplicación y al mismo tiempo ejecutar en **segundo plano** la otra actividad para continuar con la armonía de la aplicación y evitar paradas inesperadas. Es aquí donde entran los hilos, porque son los únicos que tienen la habilidad especial de permitir al programador generar concurrencia en sus aplicaciones y la sensación de multitareas ante el usuario.



Recuerda que existen dos tipos de procesamiento de tareas, **concurrencia** y **paralelismo**. El paralelismo se logra a menudo mediante el uso de múltiples núcleos de CPU, mientras que la concurrencia puede lograrse con hilos en un solo núcleo.

- La **concurrencia** se refiere a la existencia de múltiples tareas que se realizan simultáneamente compartiendo recursos de procesamiento.
- El **paralelismo** es la ejecución de varias tareas al tiempo en distintas unidades de procesamiento, por lo que es mucho más rápido que la concurrencia.

6.5.1 Tareas asíncronas

AsyncTask es una clase proporcionada por Android que facilita la ejecución de operaciones en segundo plano y la actualización de la interfaz de usuario en el hilo principal.

Sin embargo, ten en cuenta que, a partir de Android API nivel 30 (Android 11), *AsyncTask* ha quedado en desuso, y se recomienda utilizar otras alternativas.

Una tarea asíncrona, incluye los siguientes métodos que se pueden redefinir:

- onPreExecute(): se ejecuta en el hilo principal antes de iniciar la tarea en segundo plano.
- doInBackground(): se ejecuta en un hilo secundario y realiza la tarea en segundo plano.
- onProgressUpdate(): se ejecuta en el hilo principal y se llama cuando se publica el progreso durante la ejecución de la tarea en segundo plano.
- onPostExecute(): se ejecuta en el hilo principal después de que la tarea en segundo plano se complete.
- onCancelled(): se ejecuta en el hilo principal si la tarea en segundo plano se cancela.

6.5.2 Hilos con Thread y Handler

Puedes crear tus propios hilos utilizando la clase *Thread* y manejar la comunicación con el hilo principal mediante *Handler*.

```
import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    private lateinit var textView: TextView
    private lateinit var startButton: Button

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        textView = findViewById(R.id.textView)
        startButton = findViewById(R.id.startButton)

        startButton.setOnClickListener {
            // Al hacer clic en el botón, se crea un nuevo hilo con runOnUiThread
            Thread(Runnable {
                for (i in 1..5) {

                    runOnUiThread {
                        textView.text = "Iteración: $i"
                    }
                    // Simula una espera en la tarea en segundo plano
                    Thread.sleep(1000)
                }
            }).start()
        }
    }
}
```

- Se ha encapsulado el bloque de código que representa la tarea en segundo plano dentro de ***Thread(Runnable { ... }).start()***.
- Dentro del bucle de la tarea en segundo plano, se utiliza ***runOnUiThread*** para actualizar la interfaz de usuario. Esto es necesario porque las operaciones en la interfaz de usuario deben realizarse en el hilo principal de Android.

Si queremos usar también la clase *Handler*, deberíamos hacerlo así.

```
class MainActivity : AppCompatActivity() {

    private lateinit var textView: TextView
    private lateinit var startButton: Button

    // Crea un nuevo Handler asociado al Looper principal
    private val handler = Handler(Looper.getMainLooper())

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        textView = findViewById(R.id.textView)
        startButton = findViewById(R.id.startButton)

        startButton.setOnClickListener {
            // Al hacer clic en el botón, se inicia un nuevo
            Thread {
                for (i in 1..5) {
                    // Envía un mensaje al Handler para actualizar
                    handler.post {
                        textView.text = "Iteración: $i"
                    }
                    // Simula una espera en la tarea en segundo plano
                    Thread.sleep(1000)
                }
            }.start()
        }
    }
}
```

Las diferencias clave entre *handler.post* y *runOnUiThread* son:

Propósito de Uso

- runOnUiThread. Es un método conveniente proporcionado por la clase Activity y View que te permite ejecutar un bloque de código en el hilo principal.
- handler.post. Es una forma más general de publicar un mensaje en la cola de mensajes de un Handler. Puedes utilizarlo para ejecutar cualquier operación en el hilo principal o en cualquier otro hilo, dependiendo del Looper asociado al Handler.

Ubicación del Código

- runOnUiThread. Se llama directamente desde una actividad o una vista, ya que es un método de esas clases. Es específico de la interfaz de usuario de Android.
- handler.post. Puede utilizarse en cualquier parte de tu código, independientemente de las clases de Android. Puedes tener una instancia de Handler asociada a un Looper específico y usarla para comunicarte con ese hilo.

Creación del Handler

- runOnUiThread. No requiere que crees explícitamente un objeto Handler. Es proporcionado por la clase Activity o View.
- handler.post. Necesitas crear un objeto Handler y asociarlo con un Looper. Puedes usar el Looper principal para interactuar con el hilo principal.

Flexibilidad

- runOnUiThread. Es más simple y directo para realizar operaciones en el hilo principal, pero es menos flexible en comparación con Handler en términos de cómo puedes configurar y personalizar el manejo de mensajes.
- handler.post. Ofrece más flexibilidad porque puedes trabajar con cualquier Looper, ya sea el principal o uno creado por ti.

6.5.3 Corrutinas (Coroutines)

Las corrutinas son una característica importante en el lenguaje de programación **Kotlin** que facilita la escritura de **código asíncrono** y **concurrente** de manera más sencilla y eficiente que las alternativas tradicionales basadas en hilos.

Las corrutinas en Kotlin son gestionadas por la biblioteca **kotlinx.coroutines**. Proporcionan un enfoque ligero y más eficiente para la concurrencia que los hilos tradicionales, ya que permiten la suspensión de ejecución sin bloquear los hilos subyacentes.

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")  
}
```

Principales características de las corrutinas en Kotlin

- Simplicidad sintáctica
 - El código con corrutinas suele ser más claro y fácil de entender en comparación con el código basado en hilos tradicionales.
- No bloqueo
 - Las corrutinas permiten la ejecución asíncrona sin bloquear los hilos, lo que facilita la gestión de concurrencia sin sufrir los problemas asociados con el bloqueo de hilos.
- Suspensión y Continuación
 - En lugar de bloquear el hilo, las corrutinas pueden suspenderse y luego reanudarse más tarde. Esto se logra mediante palabras clave como *suspend* y funciones especiales como *launch* y *async*.
- Corrutinas en funciones asíncronas
 - Puedes usar corrutinas para escribir código asíncrono de manera secuencial, lo que mejora la legibilidad del código.

Creación de una corrutina usando el builder launch

El constructor *launch* pertenece a la interfaz *CoroutineScope* y es utilizado para lanzar una nueva corrutina. Puedes usarlo para iniciar tareas asíncronas sin bloquear el hilo principal.

En la imagen siguiente, puedes ver un ejemplo muy sencillo de corrutina:

```
import kotlinx.coroutines.*

fun main() {
    // Se lanza una corrutina
    GlobalScope.launch {
        delay(1000) // Simula una operación asíncrona
        println("¡Hola desde la corrutina!")
    }

    // Se espera a que la corrutina termine (esto no bloquea el hilo principal)
    Thread.sleep(2000)
}
```

En este ejemplo, *GlobalScope.launch* se utiliza para iniciar una nueva corrutina en el ámbito global. El bloque de código dentro de *launch* se ejecutará de forma asíncrona en un hilo separado.

Es importante mencionar que *GlobalScope* se utiliza aquí para propósitos educativos, pero en una aplicación real, sería preferible crear tu propio ámbito de corrutina limitado a un ciclo de vida específico, como el de una actividad o un fragmento en el caso de aplicaciones Android.

Funciones de suspensión

Las **funciones de suspensión** (o funciones suspendidas) son una parte fundamental de las corrutinas en Kotlin.

Estas funciones **permiten pausar la ejecución de una corrutina sin bloquear el hilo** en el que se está ejecutando. Cuando se suspende una corrutina, no se bloquea el hilo, lo que significa que se puede utilizar de manera más eficiente, especialmente en entornos asíncronos como el desarrollo de aplicaciones Android.

Las funciones de suspensión se declaran usando la palabra clave *suspend* en la definición de la función. Una función suspendida puede contener llamadas a otras funciones suspendidas, y la ejecución de la corrutina se pausa hasta que la operación suspendida se completa.

Aquí tienes un ejemplo simple de una función de suspensión:

```
import kotlinx.coroutines.delay
import kotlinx.coroutines.runBlocking

suspend fun doSomething() {
    println("Start")
    delay(1000) // Suspender la corutina durante 1 segundo
    println("End")
}

fun main() = runBlocking {
    launch {
        doSomething()
    }
}
```

Dispatchers

En Kotlin, los *dispatchers* son componentes clave que determinan en qué hilo o hilo de ejecución se llevará a cabo una tarea específica.

Kotlin proporciona varios *dispatchers* predefinidos que facilitan la ejecución de corrutinas en diferentes contextos. Algunos de los dispatchers más comunes son:

1. Dispatcher.Default. Este dispatcher está diseñado para realizar operaciones intensivas en CPU. Se utiliza comúnmente para tareas que involucran cálculos pesados.
2. Dispatcher.IO. Este dispatcher está optimizado para realizar operaciones de entrada/salida (I/O), como lecturas y escrituras de archivos o solicitudes de red.
3. Dispatcher.Main. Este dispatcher se utiliza para realizar operaciones en el hilo principal de la interfaz de usuario. Es esencial cuando necesitas actualizar la interfaz de usuario después de realizar alguna tarea asíncrona en segundo plano.
4. Dispatcher.Unconfined. Este dispatcher no impone ningún hilo específico. La corrutina se ejecutará en el mismo hilo que llama a launch. Puede ser útil en situaciones específicas, pero generalmente se recomienda utilizar dispatchers específicos.

Corrutinas y objetos Job

En Kotlin, un objeto *Job* es una entidad que representa una *tarea asíncrona* que puede ser *cancelada*. Puedes utilizar un objeto *Job* para gestionar y controlar la ejecución de una corrutina.

Aquí tienes un ejemplo simple de cómo puedes usar un objeto *Job* para controlar una corrutina:

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    // Creamos un objeto Job
    val job: Job = launch {
        repeat(10) { i ->
            println("Job is running: $i")
            delay(1000)
        }
    }

    // Pausamos la ejecución principal por un tiempo
    delay(5000)

    // Cancelamos la corrutina después de 5 segundos
    job.cancel()

    // Esperamos a que la corrutina se complete
    job.join()

    println("Main program has completed")
}
```


En este ejemplo:

- Creamos un objeto **Job** utilizando la función **launch**.
- Dentro de la corrutina, hay un bucle que se ejecuta 10 veces, imprimiendo un mensaje y luego pausando la corrutina durante 1 segundo en cada iteración.
- En la ejecución principal (**runBlocking**), esperamos 5 segundos utilizando **delay**.
- Cancelamos la corrutina llamando a **job.cancel()**. Esto detendrá la ejecución de los hijos de la corrutina.
- Luego, llamamos a **job.join()** para esperar a que la corrutina se complete.

6.5.4 Progress Bar

Una **ProgressBar** en Android es un elemento de interfaz de usuario que se utiliza para mostrar el progreso de una tarea en curso. Puede ser determinado o indeterminado, según el tipo de progreso que estés mostrando.

- **Determinada (ProgressBar determinada).** Muestra el progreso de una tarea que tiene una cantidad definida de trabajo. Por ejemplo, si estás descargando un archivo y sabes cuántos bytes hay que descargar, puedes mostrar una barra de progreso determinada.
- **Indeterminada (ProgressBar indeterminada).** Utilizada cuando no conoces la duración exacta de una tarea. Se muestra como un indicador de que la tarea está en progreso sin proporcionar información sobre cuánto tiempo llevará.

Aquí tienes un ejemplo de uso de la **ProgressBar** con una **corrutina**:

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

    <ProgressBar
        android:id="@+id/progressBar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        style="?android:attr/progressBarStyleHorizontal" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Ejecutar la corrutina en el hilo principal (UI)
        GlobalScope.launch(Dispatchers.Main) {
            // Simulamos un proceso que lleva tiempo utilizando una corrutina
            realizarTareaAsincrona()
        }

        private suspend fun realizarTareaAsincrona() {
            // Mostrar la ProgressBar al inicio de la tarea
            progressBar.visibility = android.view.View.VISIBLE

            // Simular una tarea asincrona con delay
            delay(3000)

            // Ocultar la ProgressBar después de la tarea
            progressBar.visibility = android.view.View.GONE
        }
    }
}
```

6.5.5 SplashScreen

Una "*Splash Screen*" es una pantalla de inicio que se muestra durante unos segundos al iniciar una aplicación, antes de pasar a la pantalla principal de la aplicación.

La función principal de una *SplashScreen* es proporcionar a los usuarios una interfaz visual atractiva mientras la aplicación se está cargando, inicializando o realizando tareas de configuración antes de estar lista para su uso.

Generalmente muestra el **logotipo** de la aplicación, el **nombre** de la aplicación u otros elementos visuales distintivos. Su propósito es mejorar la experiencia del usuario al dar la impresión de que la aplicación se está cargando rápidamente y para brindar una apariencia más pulida y profesional.

Mientras está visible, la aplicación puede realizar operaciones de fondo, como cargar recursos, establecer conexiones de red o realizar otras tareas de inicialización necesarias. Una vez que estas tareas han sido completadas, la *SplashScreen* desaparece y la aplicación transita a la **pantalla principal**.

Es importante diseñar y utilizar *SplashScreen* de manera **efectiva** para no causar frustración al usuario. Debe aparecer por un tiempo breve y no debería dar la sensación de que la aplicación se está cargando lentamente. Además, en algunas situaciones, como en aplicaciones que se ejecutan en segundo plano, puede ser preferible evitar el uso de SplashScreen para proporcionar una experiencia de inicio más rápida.

A partir de **Android 12**, la pantalla de presentación se aplica en cada inicio en frío de la aplicación. Si no hay personalización, la aplicación mostrará el ícono y cerrará la pantalla de presentación cuando esté disponible el nuevo fotograma.

En el siguiente [enlace](#), puedes encontrar los pasos para realizar una *SplashScreen*.

En este [enlace](#), tienes la documentación oficial.

6.5.6 Descarga de imágenes → Librería Glide

Glide es una biblioteca de gestión y carga de imágenes en Android que simplifica la tarea de cargar imágenes de manera eficiente en las aplicaciones. Fue desarrollada por el equipo de ingenieros de *Bump Technologies* y posteriormente adquirida por *Google*.

Las principales ventajas de *Glide* son:

- Realiza la carga de imágenes de manera **asincrónica**, lo que significa que no bloqueará el hilo principal de la interfaz de usuario mientras las imágenes se están descargando o procesando. Esto es esencial para mantener una interfaz de usuario receptiva.
- Permite aplicar **transformaciones** a las imágenes antes de cargarlas, como redimensionamiento, recorte, rotación, y más. Esto proporciona flexibilidad para adaptar las imágenes a los requisitos específicos de la interfaz de usuario.
- Facilita la **integración** de imágenes en diferentes tipos de widgets de la interfaz de usuario, como *ImageView*, *ImageButton*, y otros.
- Es **compatible** con varios formatos de imagen, incluyendo *GIFs animados*. Puede manejar automáticamente la reproducción de GIFs y otros formatos, lo que simplifica la integración de contenido multimedia en las aplicaciones.

Toda la documentación sobre *Glide*, puedes encontrarla en el siguiente [enlace](#).

También puedes visualizar el siguiente [vídeo](#) de *Youtube*.

6.6 Notificaciones

Para crear una notificación en una aplicación de Android, puedes utilizar la clase *NotificationCompat* del paquete de compatibilidad de Android.

Aquí tienes un ejemplo básico de cómo crear y mostrar una notificación:

1. Creamos un método llamado **showNotification()** que crea una notificación, a partir de un objeto *Builder* creado desde la clase *NotificationCompat*.

```
private fun showNotification() {  
    // Crear una notificación  
    val builder = NotificationCompat.Builder(this, CHANNEL_ID)  
        .setSmallIcon(R.drawable.ic_notification)  
        .setContentTitle("Notificación de ejemplo")  
        .setContentText("¡Hola! Esto es un ejemplo de notificación en Android.")  
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)  
  
    // Mostrar la notificación  
    with(NotificationManagerCompat.from(this)) {  
        notify(1, builder.build())  
    }  
}
```

2. Creamos un método llamado ***createNotificationChannel()*** que crea una canal de notificación, necesario a partir de Android 8.0 (Oreo)

```
private fun createNotificationChannel() {  
    // Crear el canal de notificación, necesario a partir de Android 8.0 (Oreo)  
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
        val channel = NotificationChannel(  
            CHANNEL_ID,  
            "My Channel",  
            NotificationManager.IMPORTANCE_DEFAULT  
        )  
        val notificationManager =  
            getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager  
        notificationManager.createNotificationChannel(channel)  
    }  
}
```

3. Finalmente, desde el método ***onCreate()***, creamos y lanzamos la notificación, cuando se hace click desde un botón.

```
class MainActivity : AppCompatActivity() {  
  
    private val CHANNEL_ID = "my_channel"  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        createNotificationChannel()  
  
        btnShowNotification.setOnClickListener {  
            showNotification()  
        }  
    }  
}
```

Un aspecto importante, es establecer el **nivel de importancia** del canal de notificación. En la siguiente tabla, puedes ver los diferentes niveles existentes:

Tabla 1: Niveles de importancia del canal

Nivel de importancia visible para el usuario	Importancia (Android 8.0 y versiones posteriores)	Prioridad (Android 7.1 y versiones anteriores)
Urgente: Emite un sonido y aparece como una notificación de atención	IMPORTANCE_HIGH	PRIORITY_HIGH o PRIORITY_MAX
Alta: Emite un sonido	IMPORTANCE_DEFAULT	PRIORITY_DEFAULT
Media: Sin sonido	IMPORTANCE_LOW	PRIORITY_LOW
Baja: Sin sonido; no aparece en la barra de estado	IMPORTANCE_MIN	PRIORITY_MIN

6.7 Vibración

En este ejemplo básico de cómo usar el vibrador, se tiene en cuenta la compatibilidad con Android 11 y versiones anteriores.

Primero, asegúrate de agregar el permiso necesario en el archivo *AndroidManifest.xml* para acceder al servicio de vibración.

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

A continuación, este es el código necesario para trabajar con el vibrador.

```
import android.content.Context
import android.os.Build
import android.os.VibrationEffect
import android.os.Vibrator
import android.os.Build.VERSION.SDK_INT
import android.os.Build.VERSION_CODES.S
import android.os.Build.VERSION_CODES.O
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.os.VibratorManager
import android.widget.Button
import androidx.annotation.RequiresApi
import androidx.core.content.ContextCompat.getSystemService

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val fetchDataButton: Button = findViewById(R.id.fetchDataButton)

        fetchDataButton.setOnClickListener { it: View!
            // llamada a la función de vibración
            vibrarDispositivo()
        }
    }
}
```

```

fun vibrarDispositivo()
{
    //Build.VERSION_CODES.S --> Android 12 Snow Cone
    val vibrator = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        val vibratorManager = getSystemService(Context.VIBRATOR_MANAGER_SERVICE) as VibratorManager
        vibratorManager.defaultVibrator
    } else {
        @Suppress( ...names: "DEPRECATION")
        getSystemService(AppCompatActivity.VIBRATOR_SERVICE) as Vibrator
    }

    val duration = 200L
    //Build.VERSION_CODES.O --> Android 8.0 Oreo
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O)
    {
        vibrator.vibrate(VibrationEffect.createOneShot(duration, VibrationEffect.DEFAULT_AMPLITUDE))
    }
    else
    {
        @Suppress( ...names: "DEPRECATION")
        vibrator.vibrate(duration)
    }
}
}

```

- **Build.VERSION_CODES.S** (Android 12). A partir de esta versión, VIBRATOR_SERVICE está obsoleto.
- **Build.VERSION_CODES.O** (Android 8.0). Las versiones anteriores a Android 8.0, no permiten el uso de efectos de vibración.

6.8 Reproducción de sonidos

La reproducción de sonidos en Android es bastante sencilla. Tienes toda la información en el siguiente enlace:

<https://developer.android.com/guide/topics/media/mediaplayer?hl=es-419#kotlin>

Cuando trabajamos con sonido y vídeo, debemos añadir al fichero *AndroidManifest* los siguientes permisos:

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

Un ejemplo de app que reproduce un sonido sería:

```
import android.media.MediaPlayer
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    private var mediaPlayer: MediaPlayer? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Inicializar el MediaPlayer
        mediaPlayer = MediaPlayer.create(this, R.raw.sample_audio)

        // Configurar un listener para manejar eventos de finalización
        mediaPlayer?.setOnCompletionListener {
            // Acciones a realizar cuando la reproducción ha finalizado
            // Por ejemplo, detener el MediaPlayer
            stopMediaPlayer()
        }

        // Iniciar la reproducción
        playMediaPlayer()
    }
}
```



```

private fun playMediaPlayer() {
    mediaPlayer?.start() // Comienza la reproducción
}

private fun pauseMediaPlayer() {
    mediaPlayer?.pause() // Pausa la reproducción
}

private fun stopMediaPlayer() {
    mediaPlayer?.stop() // Detiene la reproducción
    mediaPlayer?.release() // Libera los recursos del MediaPlayer
    mediaPlayer = null
}

override fun onDestroy() {
    super.onDestroy()
    // Asegurarse de liberar recursos del MediaPlayer cuando la act
    stopMediaPlayer()
}
}

```

- **MediaPlayer.create(this, R.raw.sample_audio)** inicializa el MediaPlayer con el archivo de audio ubicado en res/raw/sample_audio.mp3.
- **setOnCompletionListener** permite configurar un listener para manejar eventos de finalización de la reproducción, como detener el MediaPlayer cuando la reproducción ha finalizado.
- Las funciones `playMediaPlayer()`, `pauseMediaPlayer()`, y `stopMediaPlayer()` se utilizan para controlar la reproducción, pausa y detención del MediaPlayer respectivamente.
- En el método **onDestroy()**, se asegura de liberar los recursos del MediaPlayer cuando la actividad se destruye para evitar posibles pérdidas de memoria.