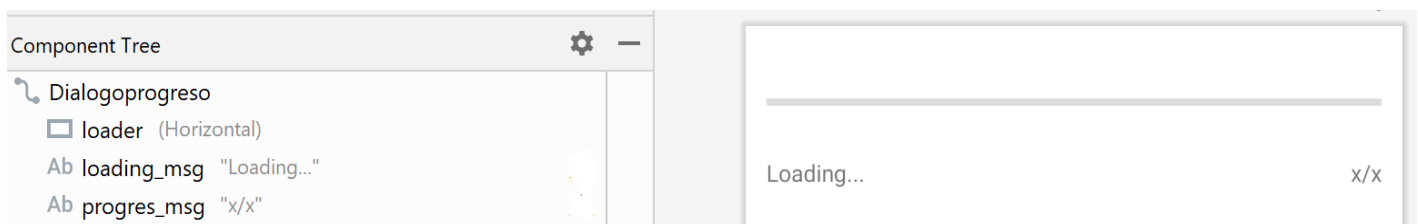


Ejercicio resuelto Progress indicators

[Descargar estos apuntes](#)

Vamos a desarrollar una aplicación que simule la carga de un conjunto de archivos, simularemos la carga mediante `AsyncTask` donde especificaremos un retardo por cada paso de la tarea, y utilizaremos una `LinearProgressIndicator` para ver la evolución del estado de la carga. Todo el proceso comenzará cuando pulsemos el botón de `Load` en la pantalla principal.

El diseño de nuestro `LinearProgressIndicator` va a ser personalizado, incluyendo unos campos de texto tal y como se ve en la siguiente imagen:



El archivo xml correspondiente a esta interfaz sería:

```

<?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:padding="13dp"
    android:id="@+id/Dialogoprogreso"
    android:layout_centerHorizontal="true"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    12 <com.google.android.material.progressindicator.LinearProgressIndicator
        android:id="@+id/loader"
        style="@style/Widget.MaterialComponents.LinearProgressIndicator"
        android:layout_width="match_parent"
        android:layout_height="65dp"
        app:layout_constraintStart_toStartOf="parent"
    18 app:layout_constraintTop_toTopOf="parent"/>

    20 <TextView
        android:id="@+id/loading_msg"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:text="Loading..."
        android:textAppearance="?android:textAppearanceSmall"
        app:layout_constraintStart_toStartOf="parent"
    28 app:layout_constraintTop_toBottomOf="@id/loader"/>

    30 <TextView
        android:id="@+id/progres_msg"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="x/x"
        android:textAppearance="?android:textAppearanceSmall"
        app:layout_constraintEnd_toEndOf="parent"
    37 app:layout_constraintTop_toBottomOf="@+id/loader" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

Aclaraciones:

- **Líneas 12-18:** definición del **LinearProgressIndicator** . En la línea 16 especificamos el texto que se visualizará cuando el **FAB** esté pulsado. El resto de propiedades ya están explicadas.
- **Líneas 20-28:** aquí definimos el primer **Text** que indicará lo que hace nuestra **LinearProgressIndicator**
- **Líneas 30-37:** aquí definimos el texto asociado al progreso de la acción (1/10, 2/10...)

Tareas asíncronas

Un hilo es una unidad de ejecución asociada a una aplicación. Es la estructura de programación concurrente que tiene como objetivo dar la percepción al usuario que el sistema ejecuta múltiples tareas a la vez.

Desde **Android** podemos ejecutar tareas en segundo plano utilizando **thread** (hilos), con **AsyncTask** y con corrutinas **Kotlin**.

El problema con los hilos es que no podemos acceder a ningún elemento de la interfaz gráfica desde dentro del mismo.

La finalidad de **AsyncTask** es unificar la actualización de la vista asociada a la tarea en segundo plano con la ejecución de la misma. Está marcada como **deprecated** desde la versión 11 de **Android (API 30)**. A partir de esta versión se recomienda el uso de corrutinas.

AsyncTask es una interfaz que nos va a permitir crear un hilo secundario en el que realizar tareas en background.

La definición genérica de una tarea asíncrona sigue el siguiente esquema:

```
class MiTarea:AsyncTask<T1,T2,T3>(){
    override fun onPreExecute(){
        //...
    }

    override fun doInBackground(vararg params:T1):T3{
        //...
        return T3
    }

    override fun onProgressUpdate(vararg params:T2){
        //...
    }

    override fun onPostExecute(result:T3){
        //...
    }

    override fun onCancelled(){
        //...
    }
}
```

El método encargado de ejecutar el hilo secundario es **doInBackground()**, el resto se ejecutan en el hilo principal.

El método `onPreExecuted()` se ejecuta antes de empezar la tarea en segundo plano.

En la definición se especifican tres tipos de datos:

- El primero es el que recibe el método `doInBackground()`. La notación `vararg params:T1` indica que se pueden recibir un número indeterminado de parámetros indeterminado del tipo `T1`.
- El segundo es el que permite comunicar el avance de la tarea. La comunicación con el hilo principal se realizará a través de la invocación del método `publishProgress()` que hará que se ejecute el método `onProgressUpdate()` que se encargará de la actualización de la interfaz.
- El tercero es el tipo de datos que se devolverá al finalizar el hilo secundario `doInBackground()` y que será recibido en `onPostExecute()`.

El método `onCanceled()` se ejecuta si la tarea es cancelada, y hace que no se ejecute el `onPostExecute()`.

En general se recomienda trabajar con las corrutinas frente a `AsyncTask` y `Thread`, son más sencillas de utilizar y generan un código más claro al interactuar con la interfaz.

Es importante, sin embargo, conocer como funcionan, sobre todo `AsyncTask`, ya que encontraremos multitud de ejemplos de código implementados con esta interfaz.

Corrutinas Kotlin

Para comenzar a utilizarlas lo primero es añadir en el fichero `build.gradle` perteneciente a `Module:app`, las siguientes referencias:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9'
```

Existen diferentes formas de trabajar con corrutinas. En este punto vamos a ver cómo hacerlo con `CoroutineScope`. Para lanzar una tarea en segundo plano mediante esta clase usaremos su método `launch`. En este método se especifica el `dispatcher` (emisor o tipo de hilo) donde tiene que ejecutarse el código. Kotlin proporciona tres `dispatcher`:

- `Dispatcher.Default` el emisor por defecto de las tareas en segundo plano.
- `Dispatcher.IO`, hilo secundario para tareas de entrada/salida, permite paralelizar más tareas concurrentes que el anterior.
- `Dispatcher.Main`, donde se indica el código que se ejecuta en el hilo principal y permite, por tanto, interactuar con la interfaz de usuario.

Veamos el código que permite visualizar una barra de progreso mientras se cargan unos archivos (de forma simulada 😊)

```

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

    binding.buttonProgress.setOnClickListener {
        8         setDialog()
        9         GlobalScope.launch (Dispatchers.IO){
        10             for (i in 0..100) {
        11                 Thread.sleep(100)
        12                 launch(Dispatchers.Main) {
                    progressBar!!.progress=i
                    progressMessage.text= i.toString()+"/100"
                }
            }
        17         launch(Dispatchers.Main){
            Toast.makeText(this@MainActivity, "Tarea finalizada!", Toast.LENGTH_S
            dialog.dismiss()
        }
    }
}


private fun setDialog() {
    val builder = MaterialAlertDialogBuilder(this)
    val inflater = this@MainActivity.layoutInflater
    val v = inflater.inflate(R.layout.dialogo_progress, null)
    progressBar = v.findViewById(R.id.loader)
    loadingMessage = v.findViewById(R.id.loading_msg)
    progressMessage = v.findViewById(R.id.progres_msg)
    builder.setView(v)
    dialog = builder.create()
    dialog.setCancelable(false)
    dialog.show()
}
}

```

Aclaraciones:

- **Línea 8**, ejecutamos en hilo principal y tras la pulsación del botón, el diálogo correspondiente a nuestra **bara de progreso**.
- **Línea 9**, invocamos a `GlobalScope.launch (Dispatchers.IO)` el hilo secundario. Simularemos un proceso de carga de archivos (**línea 10**) mediante un `sleep` (**línea 11**).
- **Línea 12**, ejecutamos código destinado a actualizar la interfaz del hilo principal con `launch(Dispatchers.Main)`, concretamente actualizamos la `LinearProgressIndicatos` y las vistas de texto.

- **Línea 17**, cuando terminamos la tarea secundaria de carga de archivos, cerramos el diálogo, para ello volvemos a envolver el código correspondiente en un `launch(Dispatchers.Main)` .

 **Ejercicio propuesto:** Si en el ejemplo desarrollado anteriormente giramos la pantalla ¿qué sucede?. Plantear una solución a esta problemática.