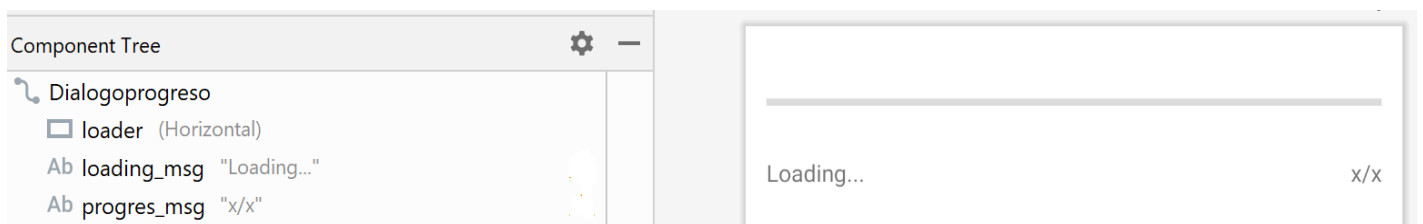


Exercici resol Progress indicators

[Descarregar aquests apunts](#)

Desenvoluparem una aplicació que simule la càrrega d'un conjunt d'arxius, simulem la càrrega mitjançant **AsyncTask** on especificarem un retard per cada pas de la tasca, i utilitzarem una **LinearProgressIndicator** per a veure l'evolució de l'estat de la càrrega. Tot el procés començarà quan premem el botó de **Load** en la pantalla principal.

El disseny del nostre **LinearProgressIndicator** serà personalitzat, incloent uns camps de text tal com es veu en la següent imatge:



L'arxiu xml corresponent a aquesta interfície seria:

```

<?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>

```

✧ Aclariments:

- **Línies 12-18:** definició del **LinearProgressIndicator** . En la línia 16 especifiquem el text que es visualitzarà quan el **FAB** estiga premut. La resta de propietats ja estan explicades.
- **Línies 20-28:** ací definim el primer **Text** que indicarà el que fa la nostra **LinearProgressIndicator**
- **Línies 30-37:** ací definim el text associat al progrés de l'acció (1/10, 2/10...)

Tasques asíncrones

Un fil és una unitat d'execució associada a una aplicació. És l'estructura de programació concurrent que té com a objectiu donar la percepció a l'usuari que el sistema executa múltiples tasques alhora.

Des de **Android** podem executar tasques en segon pla utilitzant `thread` (fils), amb `AsyncTask` i amb corrutines `Kotlin`.

El problema amb els fils és que no podem accedir a cap element de la interfície gràfica des de dins d'aquest.

La finalitat de `AsyncTask` és unificar l'actualització de la vista associada a la tasca en segon pla amb l'execució d'aquesta. Està marcada com `deprecated` des de la versió 11 de **Android (API 30)**. A partir d'aquesta versió es recomana l'ús de corrutines.

`AsyncTask` és una interfície que ens permetrà crear un fil secundari en el qual fer tasques en background.

La definició genèrica d'una tarea asíncrona segueix el següent 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 mètode encarregat d'executar el fil secundari és `doInBackground()`, la resta s'executen en el fil principal.

El mètode `onPreExcuted()` s'executa abans de començar la tasca en segon pla.

En la definició s'especifiquen tres tipus de dades:

- El primer és el que rep el mètode `doInBackground()`. La notació `vararg params:T1` indica que es poden rebre un nombre indeterminat de paràmetres indeterminat del tipus `T1`.
- El segon és el que permet comunicar l'avanç de la tasca. La comunicació amb el fil principal es realitzarà a través de la invocació del mètode `publicProgress()` que farà que s'execute el mètode `onProgressUpdate()` que s'encarregarà de l'actualització de la interfície.
- El tercer és el tipus de dades que es retornarà en finalitzar el fil secundari `doInBackground()` i que serà rebut en `onPostExecute()`.

El mètode `onCanceled()` s'executa si la tasca és cancel·lada, i fa que no s'execute el `onPostExecute()`.

En general es recomana treballar amb les corrutines enfront de `AsyncTask` i `Thread`, són més senzilles d'utilitzar i generen un codi més clar en interactuar amb la interfície.

És important, no obstant això, conèixer com funcionen, sobretot `AsyncTask`, ja que trobarem multitud d'exemples de codi implementats amb aquesta interfície.

Corrutines Kotlin

Per a començar a utilitzar-les el primer és afegir en el fitxer `build.gradle` pertanyent a `Module:app`, les següents referències:

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

Existeixen diferents maneres de treballar amb corrutines. En aquest punt veurem com fer-ho amb `CoroutineScope`. Per a llançar una tasca en segon pla mitjançant aquesta classe usarem el seu mètode `launch`. En aquest mètode s'especifica el `dispatcher` (emissor o tipus de fil) on ha d'executar-se el codi. Kotlin proporciona tres `dispatcher`:

- `Dispatcher.Default` l'emissor per defecte de les tasques en segon pla.
- `Dispatcher.io`, fil secundari per a tasques d'entrada/eixida, permet *paralelitzar més tasques concurrents que l'anterior.
- `Dispatcher.Main`, on s'indica el codi que s'executa en el fil principal i permet, per tant, interactuar amb la interfície d'usuari.

Vegem el codi que permet visualitzar una barra de progrés mentre es carreguen uns arxius (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()
}
}

```

Aclariments:

- **Línia 8**, executem en fil principal i després de la pulsació del botó, el diàleg corresponent a la nostra **bara de progrés**.
- **Línia 9**, invoquem a **GlobalScope.launch (Dispatchers.io)** el fil secundari. Simularem un procés de càrrega de arxius (**línia 10**) mitjançant un **sleep** (**línia 11**).
- **Línia 12**, executem codi destinat a actualitzar la interfície del fil principal amb **launch(Dispatchers.Main)** , concretament actualitzem la **LinearProgressIndicatos** i les vistes de text.

- **Línia 17**, quan acabem la tasca secundària de càrrega d'arxius, tanquem el diàleg, per a això tornem a embolicar el codi corresponent en un **launch(Dispatchers.Main)** .

 **Exercici proposat:** Si en l'exemple desenvolupat anteriorment girem la pantalla què succeeix?. Plantejar una solució a aquesta problemàtica.