Ejecución en el background

Ahora que sabemos un poco sobre los elementos y las pantallas de la IU, debemos hacer que sean responsivos. La capacidad de responsividad no se trata solo de velocidad - cuánto trabajo puedes hacer en un período de tiempo. De lo que se trata más es de lo rápido que se siente la aplicación.

Cuando las personas dicen que una aplicación es responsiva, lo que quieren decir con frecuencia es que la aplicación no les impide hacer lo que intentan hacer. No se interpone en su camino. Si alguna vez has usado una aplicación que simplemente se congela cuando se hace clic en un botón determinado, puedes apreciar de lo que estamos hablando. No bloquea.

Piensas en bloquear como llamar a alguien por teléfono. Cuando marcas, escuchas el timbre y esperas a que la otra persona conteste. A menos que la otra persona conteste, la llamada no puede continuar. Podemos decir que una llamada telefónica es una operación de bloqueo porque las cosas tienen que suceder en secuencia.

Marcas, suena el teléfono, la otra persona lo toma, luego hablas. Ninguna de estas cosas puede suceder al mismo tiempo. Todos los pasos implican alguna forma de “espera” o, en la terminología computacional, de bloqueo.

En este tema, analizaremos qué sucede cuando algunas tareas tardan mucho en finalizar y lo que podemos hacer para evitar estos problemas.

Tareas de larga duración

Los usuarios podrían tolerar el bloqueo en su vida cotidiana, como hacer fila para renovar licencias o comprar alimentos, o esperar a que alguien descuelgue el teléfono, y así sucesivamente. Pero pueden ser menos tolerantes al usar una aplicación. Incluso la plataforma Android no tolerará tu aplicación si toma demasiado tiempo hacer lo que sea que estés haciendo: el WindowManager y el ActivityManager de Android son los policías que responden.

Cuando un usuario hace clic en un botón o interactúa con cualquier vista que desencadena un evento, su aplicación no tiene mucho tiempo para terminar lo que se supone que debe hacer; de hecho, tiene como máximo 5 segundos antes de que muera por el tiempo de ejecución. Y para entonces, verás el infame ANR error (la aplicación no responde). Piensa en ello como el BSOD de Android (pantalla azul de la muerte).

De acuerdo con las directrices de Android, una aplicación tiene entre 100 ms y 200 ms para completar una tarea en un controlador de eventos: no es mucho tiempo, así que realmente debemos asegurarnos de no hacer nada demasiado alocado dentro de un controlador de eventos.

Pero eso es más fácil decirlo que hacerlo, y hay un par de escenarios en los que no controlaremos por completo las cosas que hacemos dentro de un controlador de eventos. Podemos enumerar algunos de ellos aquí

- Cuando leemos un archivo - Nuestros programas necesitan guardar datos o leerlos en algún momento. La operación del archivo IO puede ser notoriamente impredecible a veces; simplemente no sabes cuán grande será ese archivo. Si es demasiado grande, puede llevarse más de 200ms para completar las tareas.

- Cuando interactuamos con una base de datos - Interactuamos con una base de datos al darle comandos para leer, actualizar, crear y borrar datos. Al igual que los archivos, a veces, podemos emitir un comando que devolverá muchos datos; nos puede llevar un tiempo procesar estos registros.

- Cuando interactuamos con la red - Cuando recibimos datos dentro y fuera de los sockets de red, estamos a merced de la condición de la red. Si no está congestionada o caída, eso es bueno para nosotros. Pero no siempre es así y no siempre es rápido; si escribes códigos que tratan con la red dentro de un controlador de eventos, corres el riesgo de ANR.

- Cuando utilizamos el código de otras personas - Confiamos cada vez más en las APIs para construir nuestras aplicaciones, y por una buena razón: nos ahorran tiempo. Pero no siempre podemos saber cómo se crean estas APIs y qué tipo de operaciones tienen bajo la capa (¿realmente siempre lees el código fuente de todas las APIs que usas?).

Entonces, ¿qué deberíamos hacer para que nuestras aplicaciones no lo hagan? se encuentren con un ANR? Ciertamente no podemos evitar las cosas enumeradas anteriormente porque las aplicaciones más modernas (y útiles) necesitarán hacer una o más (o todas) de estas cosas.

La respuesta, como resultado, es ejecutar cosas en segundo plano. Hay un par de formas de hacerlo, pero en esta sección veremos cómo ejecutar nuestros códigos en una AsyncTask.

Proyecto Demo

Los detalles del proyecto para esta práctica son los siguientes.

Application name	**Async**
Project location	Usa el predeterminado.
Form factor	Solo teléfono y tableta.
Minimum SDK	API 26 Oreo
Type of activity	Vacío
Activity name	MainActivity (predeterminado)
Layout name	activity_main (predeterminado)

Este proyecto está destinado a romperse y tener problemas de rendimiento. Cuando el usuario hace clic en “tarea de ejecución prolongada”, simulará una tarea de ejecución larga, pero todo lo que estamos haciendo es contar de 1 a 15; cada tic de la cuenta demora 2 segundos. De hecho, mantenemos al usuario como rehén durante al menos 30 segundos, durante los cuales no puede hacer mucho más en la aplicación.

La figura 11.1 muestra cómo se verá nuestra pantalla, y el Listado 11.1 muestra la definición XML para el archivo de diseño.

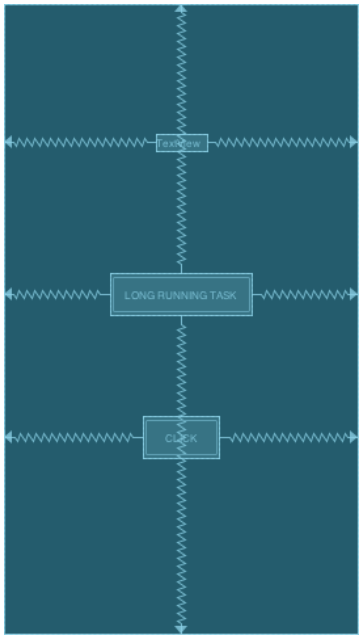


Figura 11.1: activity_main (design mode).

```xml
<?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">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="317dp"
        android:gravity="center"
        android:text="Long Running Task"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toBottomOf="@id/textView"/>

    <TextView
        android:id="@+id/textView"
        android:layout_width="184dp"
        android:layout_height="0dp"
        android:layout_marginBottom="55dp"
        android:layout_marginTop="34dp"
        android:gravity="center"
        android:text="TextView"
        android:textSize="18sp"
        app:layout_constraintBottom_toTopOf="@id/button"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Listado 11.1: activity_main.xml.

```java
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    private String TAG;
    TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button b = (Button) findViewById(R.id.button);
        Button b2 = (Button) findViewById(R.id.button2);
        tv = (TextView) findViewById(R.id.textView);
        TAG = getClass().getSimpleName();

        b.setOnClickListener(this);
        b2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.i(TAG, msg: "Clicked");
            }
        });
    }

    public void onClick(View v) { 1
        int i = 0;
        while (i < 15) {
            try {
                Thread.sleep( millis: 2000); 2
                tv.setText(String.format("Value of i = %d", i)); 3
                Log.i(TAG, String.format("Value of i = %d", i++));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Listado 11.2: MainActivity.

❶ Este bloque de código completo está diseñado para simular una actividad que consume tiempo dentro de un controlador de eventos.

❷ Esto detendrá la ejecución durante 10 segundos.

❸ Cada 10 segundos, escribimos el valor de i en la interfaz de usuario.

Este código no llegará muy lejos. Pronto encontrará un error ANR (figura 11.2) si haces clic en el botón “ejecución larga” y luego haces clic en el otro botón.

Notarás que no podrás hacer clic en él porque el subproceso de la interfaz de usuario está esperando que finalice la “tarea de larga ejecución”: la interfaz de usuario ya no responde.

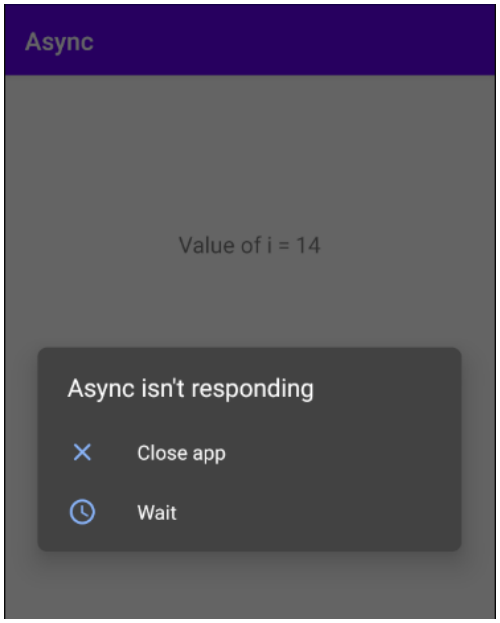


Figura 11.2: Error ANR.

AsyncTask

En la sección anterior, el problema que encontramos fue que cuando un controlador de eventos hace algo largo, toda la interfaz de usuario se congela y el usuario no puede hacer mucho más: el usuario está bloqueado.

AsyncTask estaba destinada a resolver este tipo de problemas. Fue diseñada para hacer que la IU responda, incluso cuando se realizan operaciones que llevan bastante tiempo.

La figura 11.3 muestra el papel de AsyncTask en esta solución.

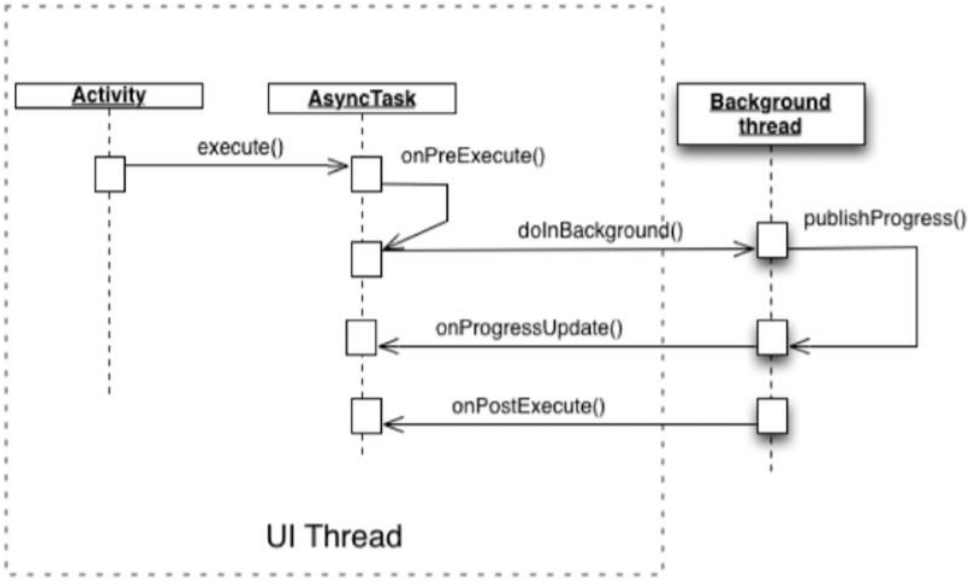


Figura 11.3: AsyncTask y MainActivty.

Esto es lo que sucede en este enfoque.

1. MainActivity crea un objeto AsyncTask (básicamente creamos una clase que extiende una AsyncTask).

2. Llama al método de ejecución de AsyncTask; en este método, pasaremos a AsyncTask las referencias de objetos del elemento UI que queremos actualizar.

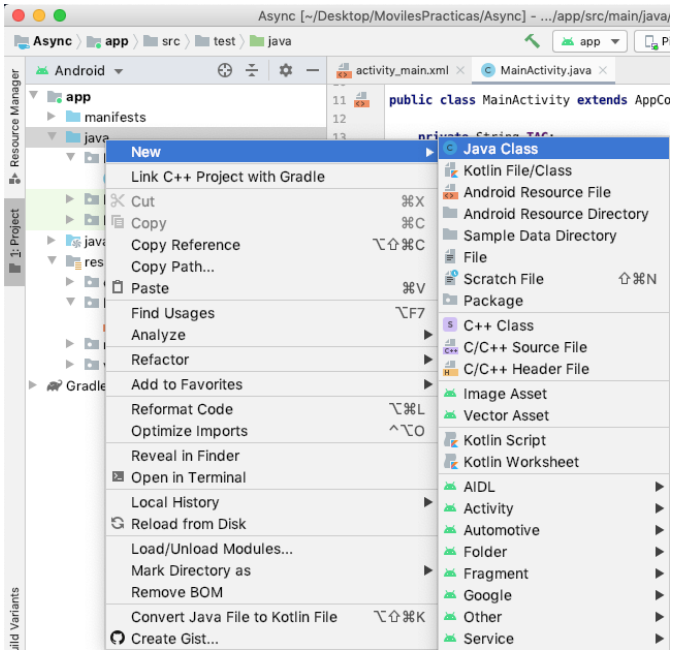
3. Hay varios métodos de ciclo de vida de AsyncTask, pero la única devolución de llamada obligatoria a anular es doInBackground() - escribiremos todas las operaciones largas aquí.

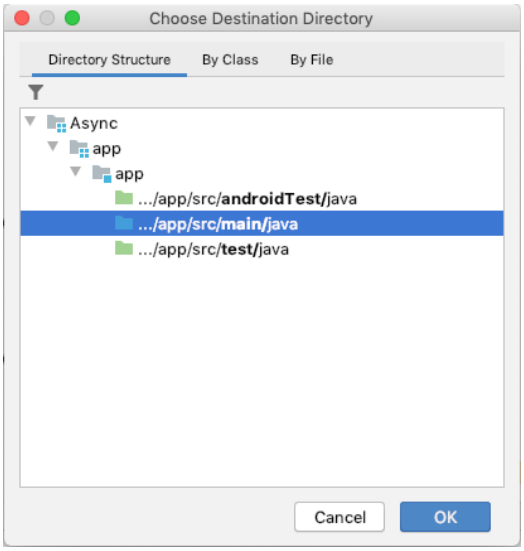
4. En este punto, AsyncTask creará un hilo de background, pero este hilo es transparente para nosotros; no nos importa porque AsyncTask será quien lo administre, no nosotros.

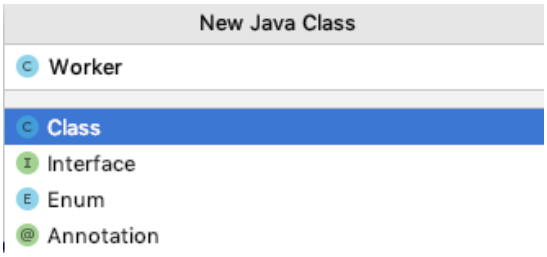
Dentro del método doInBackground(), podemos llamar periódicamente a publishProgress(). Cada vez que lo hagamos, el tiempo de ejecución llamará al método onProgressUpdate() de AsyncTask, y se realizará de manera segura para los hilos. Es dentro de este método que podemos hacer algunas actualizaciones de la IU.

Nota. Un hilo es una secuencia de instrucciones muy parecida a la secuencia de instrucciones que hemos estado escribiendo dentro de los métodos. Sin embargo, un hilo se ejecuta de forma especial: se ejecuta en segundo plano para que no bloquee lo que se está ejecutando

en el primer plano (el hilo de la interfaz de usuario). Esta es la razón por la cual necesitamos escribir instrucciones que tarden mucho tiempo en terminar en hilos.
Revisemos el proyecto AsyncTask. En primer lugar, necesitamos crear una nueva clase que se extienda desde AsyncTask.







```
package bancho.com.async;

import android.os.AsyncTask;
import android.widget.TextView;

public class Worker extends AsyncTask<TextView, String, Boolean> { 1

    private String TAG;
    private TextView tv;

    @Override
    protected Boolean doInBackground(){} 2

    @Override
    protected void onProgressUpdate(){} 3
}
```

Listado 11.3: Worker.java (Shell).

❶ The AsyncTask está parametrizada; es un tipo genérico, por lo que debemos pasarle argumentos. Estos parámetros son <Params, Progress, Result>; ver la Tabla 11.1 para más información.
❷ Este es el único método que estamos obligados a anular. Dentro de esto es donde debemos poner la lógica del programa, lo que puede tomar algún tiempo para completar.
❸ Usa este método para comunicar el progreso al usuario.

Parámetro	Descripción
1er arg (Params)	Qué información quieres pasar al hilo del fondo? Este suele ser el elemento(s) de la IU que deseas actualizar. Cuando llames a ejecutar desde MainActivity, tendrás que pasar este parámetro a AsyncTask. Este parámetro automáticamente hace su camino al método doInBackground. En nuestro ejemplo, este es un objeto de vista de texto. Queremos que el hilo del fondo tenga acceso a este elemento de la IU mientras hace su trabajo.
2º arg (Progress)	Qué tipo de información deseas que el hilo dle fondo devuelva al método ongressUpdate para que puedas especificar el estado de una operación de larga ejecución para el usuario? En nuestro caso, queremos actualizar el atributo de texto de la vista de texto, por lo que este es un objeto String.
3er arg (Result)	Qué tipo de datos deseas utilizar para especificar el estado de doInBackground cuando finaliza la tarea? En nuestro caso, solo querías que fuera verdadero si todo iba bien, por lo que el tercer parámetro es un booleano.

Tabla 11.1: Argumentos a la clase AsyncTask.

A continuación, veamos el siguiente listado.

```
package bancho.com.async;

import android.os.AsyncTask;
import android.widget.TextView;
import android.util.Log;

public class Worker extends AsyncTask<TextView, String, Boolean> {

    private String TAG;
    private TextView tv;  (1)

    @Override
    protected Boolean doInBackground(TextView... textViews){
        tv = textViews[0];  (2)
        TAG = getClass().getSimpleName();
        int i = 0;
        while (i++ < 15) {
            try {
                Thread.sleep( millis: 2000);
                publishProgress(String.format("Value of i = %d", i));  (3)
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return true;
    }

    @Override
    protected void onProgressUpdate(String... values){...}
}
```

Listado 11.4: Clase Worker Shell.

❶ La vista de texto está declarada en la parte superior de la clase, así que podemos acceder a ella desde onProgressUpdate; aún no podemos definirla porque solo obtendremos referencia de objeto a esta vista de texto cuando se llame a doInBackground.

❷ Ahora podemos definir la vista de texto; ya se nos pasó cuando MainActivity llamó al método execute().

El parámetro de este método es una matriz, pero sabemos que solo pasamos un objeto IU (la vista de texto), por lo que obtenemos solo el primer elemento de la matriz. Ahora podemos almacenar esa referencia a la variable TextView (tv) que levantamos en ❶.

❸ En cada tick, llamaremos a publishProgress, por lo que puede actualizar la IU.

A continuación, implementemos el método onProgressUpdate.

```
package bancho.com.async;

import android.os.AsyncTask;
import android.widget.TextView;
import android.util.Log;

public class Worker extends AsyncTask<TextView, String, Boolean> {

    private String TAG;
    private TextView tv;

    @Override
    protected Boolean doInBackground(TextView... textViews){...}

    @Override
    protected void onProgressUpdate(String... values){
        tv.setText(values[0]);
        Log.i(TAG, String.format(values[0]));
    }
}
```

Listado 11.5: onProgressUpdate.

Este método capturará los valores que pasamos al método publishProgress. El parámetro de este método es, nuevamente, una matriz.

Y dado que solo le pasamos una cadena, solo obtendremos el primer elemento y estableceremos su valor como el atributo de texto del objeto de vista de texto.

Básicamente hemos reubicado la tarea que consume mucho tiempo en MainActivity y la colocamos dentro de la clase Worker. El siguiente paso es actualizar los códigos en MainActivity.

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    private String TAG;
    TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button b = (Button) findViewById(R.id.button);
        Button b2 = (Button) findViewById(R.id.button2);
        tv = (TextView) findViewById(R.id.textView);
        TAG = getClass().getSimpleName();

        b.setOnClickListener(this);
        b2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Log.i(TAG, msg: "Clicked");
            }
        });
    }

    public void onClick(View v) {
        Worker worker = new Worker();
        worker.execute(tv);
    }
}
```

Listado 11.6: MainActivity.

❶ El bloque onCreate permanece sin cambios desde la sección anterior; simplemente configuramos los controladores de eventos aquí.

❷ Crea una instancia de la clase AsyncTask Worker. Ten en cuenta que la ejecución en segundo plano de AsyncTask no se inicia simplemente creando una instancia de la misma.

❸ El método de ejecución inicia la operación de fondo. En este método, pasamos lo que queremos actualizar a AsyncTask. Ten en cuenta que puedes pasar más de un elemento IU al método de ejecución, ya que se pasará como una matriz en el método doInBackground de AsyncTask.

Nota. AsyncTask no está diseñado para ejecutar operaciones muy largas, cosas del orden de minutos. Generalmente, AsyncTask se usa solo para operaciones que duran un par de segundos. Más que eso y WindowManager/ActivityManager aún puede matar la aplicación.

Para operaciones de larga ejecución, necesitas usar Servicios, pero eso está más allá del alcance de este parte del curso.

```
package bancho.com.async;

import android.os.AsyncTask;
import android.widget.TextView;
import android.util.Log;

public class Worker extends AsyncTask<TextView, String, Boolean> {

    private String TAG;
    private TextView tv;

    @Override
    protected Boolean doInBackground(TextView... textViews){
        tv = textViews[0];
        TAG = getClass().getSimpleName();
        int i = 0;
        while (i++ < 15) {
            try {
                Thread.sleep( millis: 2000);
                publishProgress(String.format("Value of i = %d", i));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        return true;
    }

    @Override
    protected void onProgressUpdate(String... values){
        tv.setText(values[0]);
        Log.i(TAG, String.format(values[0]));
    }
}
```

Listado 11.7: Worker.

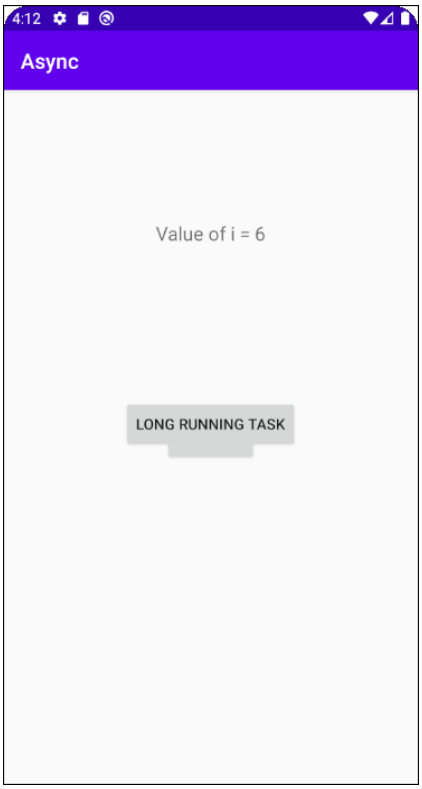


Figura 11.4: Ejecución del demo.