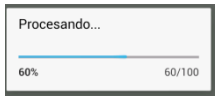


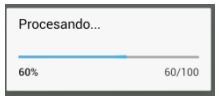
Tareas costosas

Hilos

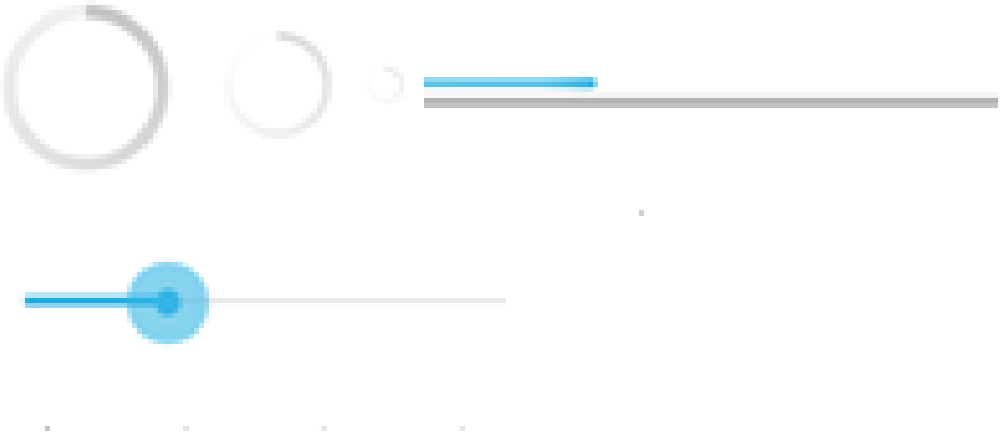
Tareas asíncronas

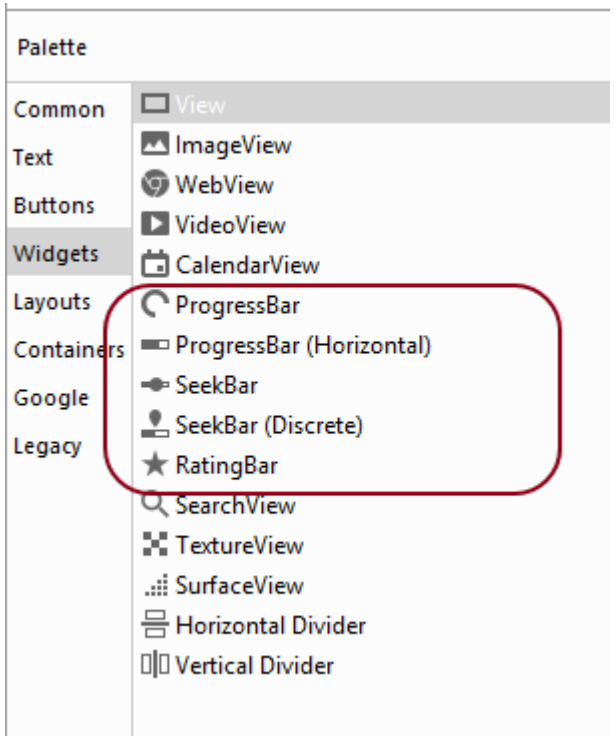
ProgressBar

Vista ProgressBar

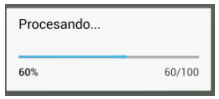


- Indicador visual de progreso en alguna operación, muy utilizado en tareas "largas" para informar al usuario.

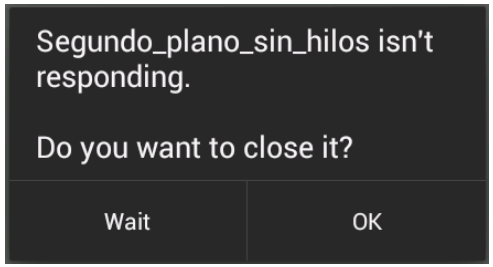




Hilo principal

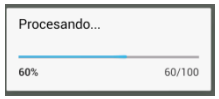


- Todos los componentes de una aplicación se ejecutan en el mismo hilo de ejecución, el llamado **hilo principal** o **hilo GUI** (éste último nombre indica también que es el hilo donde se ejecutan todas las operaciones que gestionan la interfaz gráfica de usuario de la aplicación).
- Por eso, cualquier operación larga que realicemos en este hilo va a bloquear la ejecución del resto de componentes de la aplicación y por supuesto también la interfaz, produciendo al usuario un efecto evidente de lentitud, bloqueo o mal funcionamiento en general, algo que deberíamos evitar a toda costa.

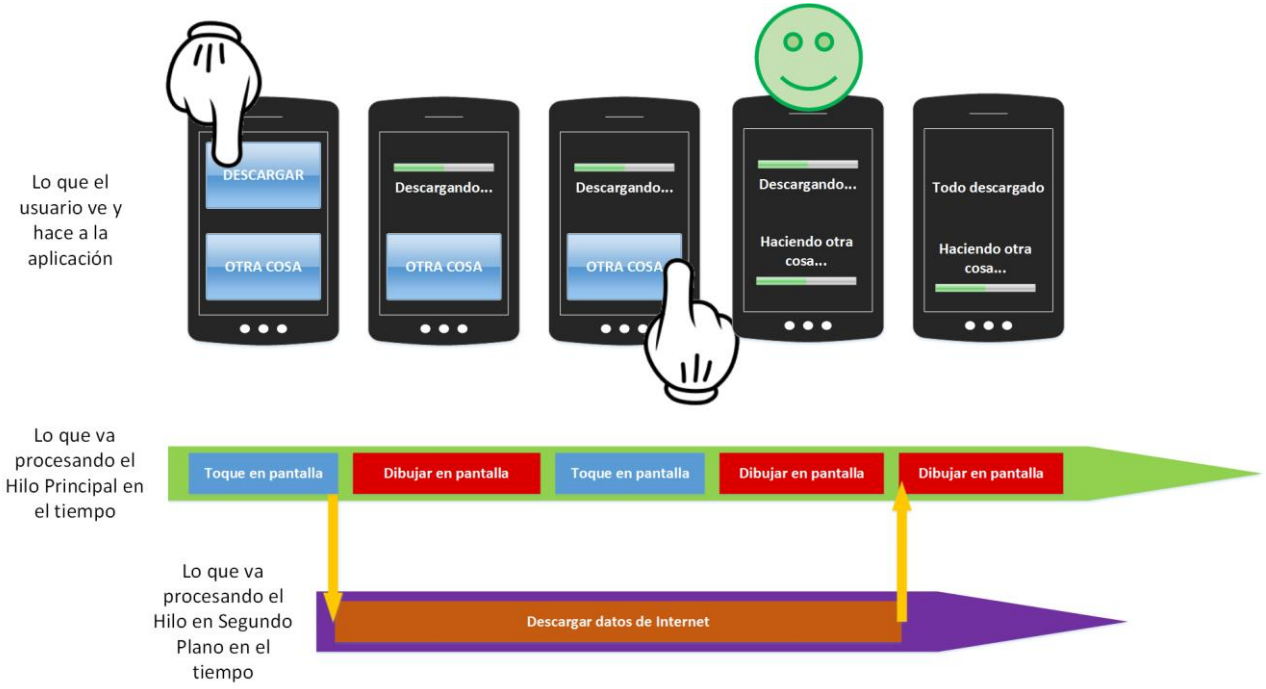


- Incluso puede ser peor, dado que Android monitoriza las operaciones realizadas en el hilo principal y puede mostrar el famoso mensaje de "*Application Not Responding*" (ANR) para que el usuario decida entre forzar el cierre de la aplicación o esperar a que termine.

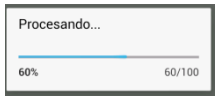
Hilos independientes



- Para evitar el problema:
  - Hilos secundarios
  - Tareas asíncronas



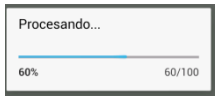
Hilos




```
new Thread (new Runnable() {
    public void run() {
        // desarrollo. . .
    }
}).start();
```

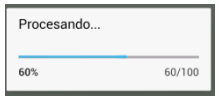
- Pero desde este hilo secundario que hemos creado no podemos hacer referencia directa a componentes que se ejecuten en el hilo principal, entre ellos los controles que forman nuestra interfaz de usuario.

Acceso desde hilo secundarios al hilo principal y a la UI



- Existen diversas maneras:
  - Activity.runOnUiThread(Runnable): para "enviar" operaciones al hilo principal desde el hilo secundario
  - View.post(Runnable): para actuar sobre cada control de la interfaz
  - View.postDelayed(Runnable, long): para actuar sobre cada control de la interfaz con un retardo temporal
  - Handler: manejador para enviar mensajes entre el hilo secundario y el principal

Ejemplo de uso de los métodos post() y runOnUiThread()



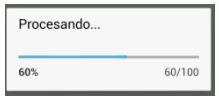

```
final ProgressBar progressBar = findViewById(R.id.progressBar);
new Thread(new Runnable() {
        public void run() {
          progressBar.post(new Runnable() {
             public void run() {
                progressBar.setProgress(0);
             }
          });
          for (int i = 1; i <= 10; i++) {
             try {
                Thread.sleep(1000);
             } catch (InterruptedException e1) {
                e1.printStackTrace();
             }
             progressBar.post(new Runnable() {
                public void run() {
                   progressBar.incrementProgressBy(10);
                }
             });
          }
          runOnUiThread(new Runnable() {
             public void run() {
                Toast.makeText(getApplicationContext(), "Tarea finalizada!", Toast.LENGTH_SHORT).show();
             }
          });
        }
}).start();
```

Actúa sobre la View de tipo ProgressBar de la interfaz gráfica y definida en el hilo principal como progressBar

"Supuesta" tarea costosa

Envía la operación Toast.make al hilo principal

Objeto Handler

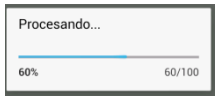



```
final Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            seekBar.incrementProgressBy(10);
        }
};
new Thread(new Runnable() {
        public void run() {
            seekBar.post(new Runnable() {
                public void run() {
                    seekBar.setProgress(0);
                }
            });
            for (int i = 1; i <= 10; i++) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e1) {
                    e1.printStackTrace();
                }
                handler.sendMessage(handler.obtainMessage());
            }
            runOnUiThread(new Runnable() {
                public void run() {
                    Toast.makeText(MainActivity.this, "Hilo 2 finalizado!", Toast.LENGTH_SHORT).show();
                }
            });
        }
}).start();
```

El hilo principal crea el manejador con la acción a realizar

El hilo secundario recibe el mensaje del manejador

Objeto Handler

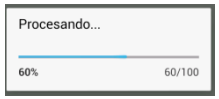



```
final SeekBar seekBar = findViewById(R.id.seekBar);
final Handler handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
           seekBar.incrementProgressBy(10);
        }
};
new Thread(new Runnable() {
        public void run() {
           seekBar.post(new Runnable() {
              public void run() {
                 seekBar.setProgress(0);
              }
           });
           for (int i = 1; i <= 10; i++) {
              try {
                 Thread.sleep(1000);
              } catch (InterruptedException e1) {
                 e1.printStackTrace();
              }
              handler.sendMessage(handler.obtainMessage());
           }
           runOnUiThread(new Runnable() {
              public void run() {
                 Toast.makeText(MainActivity.this, "Hilo 2 finalizado!", Toast.LENGTH_SHORT).show();
              }
           });
        }
}).start();
```

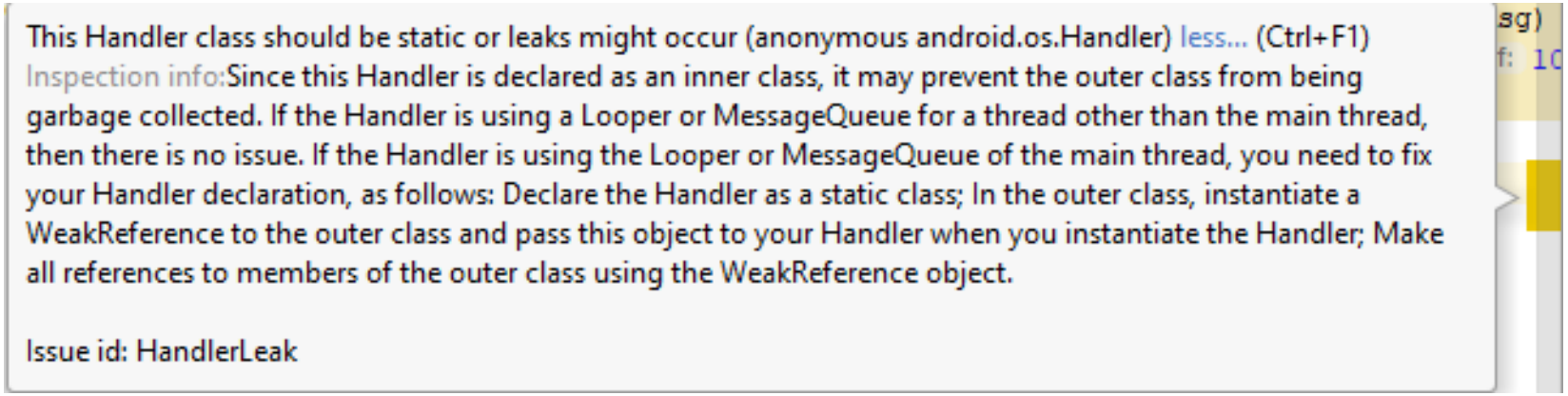
El hilo principal crea el manejador con la acción a realizar

El hilo secundario envía el mensaje del manejador

Fugas de memoria

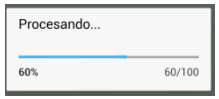


- AS advierte:



- Explicación y solución

ProgressDialog



- Subclase de AlertDialog que al utilizarla no es neceario implementar en la UI el control ProgressBar.
- Pueden hacerse cancelables al pulsar la tecla Back.

ProgressDialog

added in API level 1
Deprecated since API level 26
Summary: Constants | Inherited Constants | Ctors | Methods | Protected Methods | Inherited Methods | [Expand All]

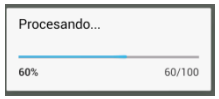
```
public class ProgressDialog
extends AlertDialog
```

java.lang.Object
↳ android.app.Dialog
↳ android.app.AlertDialog
↳ android.app.ProgressDialog

This class was deprecated in API level 26.

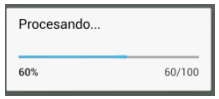
`ProgressDialog` is a modal dialog, which prevents the user from interacting with the app. Instead of using this class, you should use a progress indicator like `ProgressBar`, which can be embedded in your app's UI. Alternatively, you can use a notification to inform the user of the task's progress.

Tareas asíncronas



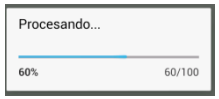
- Si el número de controles es grande, o necesitáramos una mayor interacción con la interfaz el código el uso de hilo secundario empezaría a ser inmanejable, difícil de leer y mantener y, por tanto, también más propenso a errores.
- Pues bien, la clase **AsyncTask** nos va a permitir realizar lo mismo pero con la ventaja de no tener que utilizar artefactos del tipo runOnUiThread() y de una forma mucho más organizada y legible.

Clase AsyncTask



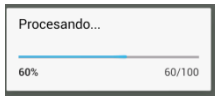
- La forma básica de utilizar la clase AsyncTask consiste en **crear una nueva clase que extienda de ella** y sobrescribir varios de sus métodos entre los que repartiremos la funcionalidad de la tarea. Estos métodos son los siguientes:
  - **onPreExecute()**: Se ejecutará antes del código principal de la tarea. Se suele utilizar para preparar la ejecución de la tarea, inicializar la interfaz, etc.
  - **doInBackground()**: Contendrá el código principal de la tarea.
  - **onProgressUpdate()**: Se ejecutará cada vez que llamemos al método **publishProgress()** desde el método doInBackground().
  - **onPostExecute()**: Se ejecutará cuando finalice nuestra tarea, es decir, tras la finalización del método doInBackground().
  - **onCancelled()**: Se ejecutará cuando se cancele la ejecución de la tarea antes de su finalización normal.
- Conexión con el hilo principal
  - **El método doInBackground() se ejecuta en un hilo secundario** (por tanto no podremos interactuar con la interfaz), pero sin embargo **todos los demás se ejecutan en el hilo principal**, lo que quiere decir que dentro de ellos podremos hacer referencia directa a nuestros controles de usuario para actualizar la interfaz.
  - Por eso, dentro de doInBackground() tendremos la posibilidad de llamar periódicamente al método publishProgress() para que automáticamente desde el método onProgressUpdate() se actualice la interfaz si es necesario.

Crear la clase extendida



- Al extender una nueva clase de AsyncTask indicaremos tres parámetros AsyncTask <TypeOfVarArgParams, ProgressValue, ResultValue>:
  1. El tipo de datos que recibiremos como entrada de la tarea en el método doInBackground().
  2. El tipo de datos con el que actualizaremos el progreso de la tarea y que recibiremos como parámetro del método onProgressUpdate() y que a su vez tendremos que incluir como parámetro del método publishProgress().
  3. El tipo de datos que devolveremos como resultado de nuestra tarea, que será el tipo de retorno del método doInBackground() y el tipo del parámetro recibido en el método onPostExecute().

Creando la clase

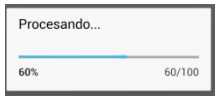



```
private class MiTareaAsincrona extends AsyncTask<Void, Integer, Boolean> {
        @Override
        protected void onPreExecute() {
                pbarProgreso.setMax(100);
                pbarProgreso.setProgress(0);
        }
        @Override
        protected Boolean doInBackground(Void... params) {
                for (int i = 1; i <= 10; i++) {
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e1) {
                                e1.printStackTrace();}
                        publishProgress(i * 10);
                        if (isCancelled())
                                break;
                }
                return true;
        }
        @Override
        protected void onProgressUpdate(Integer... values) {
                int progreso = values[0];
                pbarProgreso.setProgress(progreso);
        }
        @Override
        protected void onPostExecute(Boolean result) {
                if (result)
                        Toast.makeText(MainActivity.this, "Tarea finalizada!", Toast.LENGTH_SHORT).show();
        }
        @Override
        protected void onCancelled() {
                Toast.makeText(MainActivity.this, "Tarea cancelada!", oast.LENGTH_SHORT).show();
        }
}
```

Extendemos de AsyncTask indicando los tipos Void, Integer y Boolean respectivamente, lo que se traduce en que:

- doInBackground() no recibirá ningún parámetro de entrada (Void).
- publishProgress() y onProgressUpdate() recibirán como parámetros datos de tipo entero (Integer).
- doInBackground() devolverá como retorno un dato de tipo booleano y onPostExecute() también recibirá como parámetro un dato del dicho tipo (Boolean).

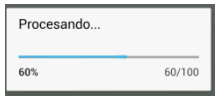
Usando la clase




```
private MiTareaAsincrona tarea1;
…
tarea1 = new MiTareaAsincrona();
tarea1.execute();
…
tarea1.cancel(true);
```

execute() porque en la definición de la tarea el primer parámetro era Void.
Si hubiera sido String deberíamos haber usado execute("qqq","qaz")

Nota sobre la Inferencia de tipos (<Tipo>)

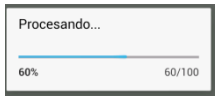


- Si nunca has visto nada de inferencia de tipos, seguramente te sorprenderá ver tipos (Integer, String, etc) entre los símbolos de "mayor que" y "menor que". Como en la cabecera de la anterior clase:

  private class MiTareaAsincrona extends AsyncTask<Void, Integer, Boolean> {

- Esto es una habilidad que tiene Java y se sirve para hacer a las variables genéricas. ¿Qué quiere decir genéricas? Que la variable puede ser del tipo que tú quieras. ¿Y por qué te puede interesar? Porque los métodos que tiene la clase usarán estas variables con el tipo que tu hayas definido, si necesitas que sea un String ¿Por qué no? Y si necesitas que alguna sea entera, pues también. La única condición es que no se permiten tipos primitivos solo tipos objetos; queremos decir con esto que para un entero no se puede poner "int", sino "Integer" –para efectos es lo mismo.

Nota sobre los argumentos variables (…)



- Puede que sea la primera vez que veas tres puntos en Java como la siguiente línea de código:
  protected void onProgressUpdate(**Integer... values**)
- Los tres puntos es casi lo mismo que un array:
- Cuando es la variable que se le pasa a una función se trabaja igual que un array normal y corriente:

```
public void hacer (String... variasCosas) {
        Log.v(TAG_LOG, "Contenido: "+ variasCosas [0] + variasCosas [1] + variasCosas [2]);
}
```

  es lo mismo que (hemos cambiado … por []):

```
public void hacer (String[] variasCosas) {
        Log.v(TAG_LOG, "Contenido: "+ variasCosas [0] + variasCosas [1] + variasCosas [2]);
}
```

- Solo cambia a la hora de llamar a la función. Si la llamamos como un array normal de toda la vida, ni lo notaremos:

```
int[] arrayDeTodaLaVida = {"Cosa 1", "Cosa 2", "Cosa 3"};
hacer (arrayDeTodaLaVida);
```

- Pero gracias a los tres puntos suspensivos, podemos meterle directamente un solo valor, directamente como:

```
hacer ("Cosa 1");
```

- O dos:

```
hacer ("Cosa 1", "Cosa 2");
```

- O tres:

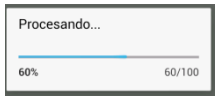
```
hacer ("Cosa 1", "Cosa 2", "Cosa 3");
```

- O ninguno:

```
hacer ();
```

- O todos los que queramos directamente, sin tener que declarar el array.
- Se puede decir que se adaptan los valores que toma la función a lo que necesitamos. Aunque lo que realmente hace es cogerlo todos juntos y meterlos en un array para después trabajar con él como ya vimos. Y como todo array tiene un tamaño (recuerdo que si llamamos a una posición del array que no existe tendremos un bonito error “java.lang Array Index Out Of Bounds Exception”), y se puede recorrer (con for, foreach, while, etc).

Fugas de memoria



- AS advierte:

This `AsyncTask` class should be static or leaks might occur (com.pdm.p_67_tarea_asincrona.MainActivity.MiTareaAsincrona) less... (Ctrl+F1)
Inspection info:A static field will leak contexts.

Non-static inner classes have an implicit reference to their outer class. If that outer class is for example a `Fragment` or `Activity`, then this reference means that the long-running handler/loader/task will hold a reference to the activity which prevents it from getting garbage collected.

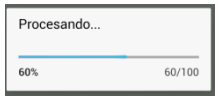
Similarly, direct field references to activities and fragments from these longer running instances can cause leaks.

ViewModel classes should never point to Views or non-application Contexts.

Issue id: StaticFieldLeak

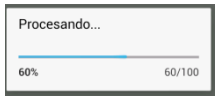
- Explicación y solución

Otras alternativas



- Otra opción es usar IntentService que no es más que un tipo particular de servicio Android que se preocupará por nosotros de la creación y gestión del nuevo hilo de ejecución y de detenerse a sí mismo una vez concluida su tarea asociada.
- Un IntentService no proporciona métodos que se ejecuten en el hilo principal de la aplicación y que podamos aprovechar para "comunicarnos" con nuestra interfaz durante la ejecución. Éste es el motivo principal de que los IntentService sean una opción menos utilizada a la hora de ejecutar tareas que requieran cierta vinculación con la interfaz de la aplicación.
- Sin embargo tampoco es imposible su uso en este tipo de escenarios ya que podremos utilizar por ejemplo mensajes *broadcast* (y por supuesto su BroadcastReceiver asociado capaz de procesar los mensajes) para comunicar eventos al hilo principal, como por ejemplo la necesidad de actualizar controles de la interfaz o simplemente para comunicar la finalización de la tarea ejecutada.

Prácticas propuestas



- Realiza la hoja de Ejercicios 15_Tareas_costosas