

Design Patterns

MSc in Computer Science

Produced
by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

<http://www.wit.ie>

<http://elearning.wit.ie>



Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRCE



Android Threads

Threads in Android

- When an application is launched, the system creates a thread of execution for the application, called "main."
- This thread is in charge of dispatching events to the appropriate user interface widgets, including drawing events.
- It is also the thread in which your application interacts with components from the Android UI toolkit (components from the `android.widget` and `android.view` packages). As such, the main thread is also sometimes called the UI thread.

Threads and UI Components

- The system does not create a separate thread for each instance of a component.
- All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread.
- Consequently, methods that respond to system callbacks (such as `onKeyDown()` to report user actions or a lifecycle callback method) always run in the UI thread of the process.

ANR Dialog

- When your app performs intensive work in response to user interaction, this single thread model can yield poor performance unless you implement your application properly.
- Specifically, if everything is happening in the UI thread, performing long operations such as network access or database queries will block the whole UI.
- When the thread is blocked, no events can be dispatched, including drawing events. From the user's perspective, the application appears to hang.
- Even worse, if the UI thread is blocked for more than a few seconds (about 5 seconds currently) the user is presented with the infamous "application not responding" (ANR) dialog.

Worker Threads

- To retain application 'responsiveness' do not block the UI thread.
- If you have operations to perform that are not instantaneous, you should make sure to do them in separate threads ("background" or "worker" threads).

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}
```

```
public void onClick(View v) {  
    new Thread(new Runnable() {  
        public void run() {  
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");  
            mImageView.setImageBitmap(b);  
        }  
    }).start();  
}
```

- At first, this seems to work fine, because it creates a new thread to handle the network operation.
- However, it violates the second rule of the single-threaded model: do not access the Android UI toolkit from outside the UI thread—this sample modifies the `ImageView` from the worker thread instead of the UI thread.
- This can result in undefined and unexpected behavior, which can be difficult and time-consuming to track down.

Android Solution

- To fix this problem, Android offers several ways to access the UI thread from other threads. Here is a list of methods that can help:
 - `Activity.runOnUiThread(Runnable)`
 - `View.post(Runnable)`
 - `View.postDelayed(Runnable, long)`

View.postDelayed(Runnable, long)

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap bitmap = loadImageFromNetwork("http://example.com/image.png")
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(bitmap);
                }
            });
        }
    }).start();
}
```

- Now this implementation is thread-safe: the network operation is done from a separate thread while the ImageView is manipulated from the UI thread.
- However, as the complexity of the operation grows, this kind of code can get complicated and difficult to maintain. To handle more complex interactions with a worker thread, you might consider using a Handler in your worker thread, to process messages delivered from the UI thread.

Async Tasks

- AsyncTask allows you to perform asynchronous work on your user interface. It performs the blocking operations in a worker thread and then publishes the results on the UI thread, without requiring you to handle threads and/or handlers yourself.
- To use it, you must subclass `AsyncTask` and implement the `doInBackground()` callback method, which runs in a pool of background threads. To update your UI, you should implement `onPostExecute()`, which delivers the result from `doInBackground()` and runs in the UI thread, so you can safely update your UI. You can then run the task by calling `execute()` from the UI thread.

```

public void onClick(View v) {
    new DownloadImageTask().execute("http://example.com/image.png");
}

private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
    /** The system calls this to perform work in a worker thread and
     * delivers it the parameters given to AsyncTask.execute() */
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

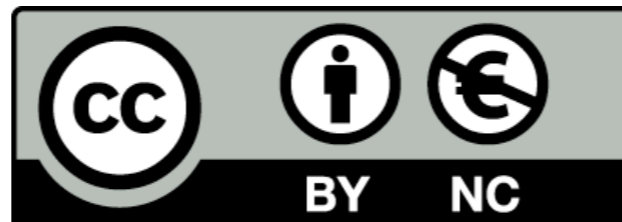
    /** The system calls this to perform work in the UI thread and delivers
     * the result from doInBackground() */
    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}

```

- Now the UI is safe and the code is simpler, because it separates the work into the part that should be done on a worker thread and the part that should be done on the UI thread.

Async Tasks

- You should read the `AsyncTask` reference for a full understanding on how to use this class, but here is a quick overview of how it works:
 - You can specify the type of the parameters, the progress values, and the final value of the task, using generics
 - The method `doInBackground()` executes automatically on a worker thread
 - `onPreExecute()`, `onPostExecute()`, and `onProgressUpdate()` are all invoked on the UI thread
 - The value returned by `doInBackground()` is sent to `onPostExecute()`
 - You can call `publishProgress()` at anytime in `doInBackground()` to execute `onProgressUpdate()` on the UI thread
 - You can cancel the task at any time, from any thread



Except where otherwise noted, this content is licensed under a Creative Commons Attribution-NonCommercial 3.0 License.

For more information, please see <http://creativecommons.org/licenses/by-nc/3.0/>

