Android Threads

Introduction

- The android platform support most of the Java SE classes. Among the ones it doesn't support we can find the UI ones.
- The android platform supports a different UI model.
- It is important to understand the android UI model in order to interact with it successfully, especially when dealing with threads issues.

The UI Thread

- Each application has a thread known as the UI thread. This thread is also called 'main'.
- The UI thread is in charge of dispatching the various events to the appropriate widgets.
- The UI thread is responsible for the drawing events and it is also the one through which the application interacts with the running components.

The UI Thread

- Having everything executed in a single thread performing long operations such as network access or database queries might block the whole user interface. From a user perspective the application will appear as if it was hung. The user might even get the 'application not responding' message.

The UI Thread

- It is highly important to keep the UI thread unblocked. When having long operations we better do them in a separated thread.

The Single Thread Rule

- Implementing a separated thread we should ensure that it doesn't violate the single threaded model.
- The android UI is not thread safe and therefore it must always be manipulated on the UI thread. Manipulating the android UI from another thread might cause unexpected results.

The `runOnUiThread(Runnable)` Method

- This method was defined as part of the `Activity` class.
- Using this method we can ensure the `run()` method will be executed as part of the UI thread.

```
public final void runOnUiThread (Runnable action)
```

If the current thread isn't the UI thread then the `run()` method will be queued in order to be performed on the UI thread when called.

The `post(Runnable)` Method

- This method was defined in `View`. It can serve us similarly to the `runOnUiThread(Runnable)` method. It ensures that the `run()` method is executed on the UI thread.

```
public boolean post(Runnable action)
```

If the current thread isn't the UI thread then the `run()` method will be queued in order to be performed on the UI thread when called.

The `post(Runnable)` Method

```
public void onClick(View v)
{
    new Thread(new Runnable()
    {
        public void run()
        {
            final Bitmap img = getImageFromServer();
            imgView.post(new Runnable()
                {
                    public void run()
                    {
                        imgView.setImageBitmap(img);
                    }
                });
        }
    }).start();
}
```

The `postDelayed(Runnable,long)` Method

- This method was defined in `View`. It can serve us similarly to the `runOnUiThread(Runnable)` method. It ensures that the `run()` method is executed on the UI thread. It sets a delay of the specified number of milliseconds.

```
public boolean post(Runnable action, long delay)
```

If the current thread isn't the UI thread then the `run()` method will be queued in order to be performed on the UI thread when called. The call to run() will be delayed in the specified number of milliseconds.

The `AsyncTask` Utility Class

- We can alternatively extend this class and override the required methods.

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

The `AsyncTask` Utility Class

```
private class DownloadTask extends AsyncTask<String, Integer, Bitmap>
{

        protected Bitmap doInBackground(String... url)
        {
            return loadFromNetwork(url);
        }

        protected void onPostExecute(Bitmap result)
        {
            imgView.setImageBitmap(result);
        }
 }
```

The `AsyncTask` Utility Class

- When declaring a class that extends `AsyncTask` we should specify three types to take the place of the following three types:

  (1) The type of the information that is needed to process the task.

  (2) The type of the information that is passed over to indicate about the progress.

  (3) The type of the information that when the task is completed is passed over to the post-task code.

The `Handler` Class

- Each Handler instance is associated with a single thread and its message queue.
- Once we get a Handler instance associated with the UI thread we can deliver Runnable objects to its message queue and have them executed as if they came out of the message queue.

The `Handler` Class

- This class is used mainly for scheduling messages and Runnable object to be executed as some point in the future and to en-queue an action to be performed on a different thread than the one we own.

The `Handler` Class

- This class has various constructors allowing us to get a new `Handler` object.

`Handler()`
Default constructor associates this handler with the queue of the current thread.

`Handler(Handler.Callback callback)`
Constructor associates this handler with the queue of the current thread and takes a callback interface in which you can handle messages.

`Handler(Looper looper)`
Use the provided queue instead of the default one.

`Handler(Looper looper, Handler.Callback callback)`
Use the provided queue instead of the default one and take a callback interface in which to handle messages.

The `Handler` Class

- Once we get a Handler object we can call various methods in order to schedule the call for `run()` method.

```
public final boolean post (Runnable r)
```

The Runnable will be executed on the thread this Handler object is associated with.

```
public final boolean postAtTime (Runnable r,
                                 long uptimeMillis)
```

The runnable will run on the thread to which this handler is attached. The runnable will run at a specific time given by uptimeMillis. The time-base is `uptimeMillis()`.

Sending Messages to Handler

- Alternatively for passing over runnable objects we can send messages.

Sending Messages to Handler

- In order to send a message we first need to obtain one. In order to obtain a message object we should call the `obtainMessage()` method on the Handler object we are working with.

```
...
Message message = handler.obtainMessage()
...
```

Sending Messages to Handler

- Sending a message is done by calling the `sendMessage()` method.

```
...
handler.sendMessage(message);
...
```

Sending Messages to Handler

- In order to process the messages our Handler should implement the `handleMessage()` method that will be called with each message that arrives to the messages queue the handler handles.

Sending Messages to Handler

- The `handleMessage()` method is called within the thread the Handler object is associated with. Assuming it is associated with the UI thread (also known as the 'main' thread) the `handleMessage()` will be called within the UI thread (the 'main' thraed).
- When instantiating Handler it is by default associate with the current thread during its instantiation.

Sending Messages to Handler

- The thread that calls the activity `onCreate()` method is the main thread, also known as the UI thread.

Sending Messages to Handler

```
public class HoloActivity extends Activity
{
    ProgressBar progressBar;
    Handler handler = new Handler()
    {
        @Override
        public void handleMessage(Message message)
        {
            progressBar.incrementProgressBy(10);
        }
    };
    boolean isRunning = false;
    @Override
    public void onCreate(Bundle bndl)
    {
        super.onCreate(bndl);
        setContentView(R.layout.main);
        progressBar = (ProgressBar) findViewById(R.id.progress);
    }
```

Sending Messages to Handler

```
public void onStart()
{
    super.onStart();
    progressBar.setProgress(0);
    Thread background = new Thread(new Runnable()
    {
        public void run()
        {
            try
            {
                for (int i = 0; i < 10 && isRunning; i++)
                {
                    Thread.sleep(1000);
                    handler.sendMessage(handler.obtainMessage());
                    Log.i("msg","i="+i);
                }
            }
            catch (Throwable t)
            {
                // end the thread
            }
        }
    });
```

Sending Messages to Handler

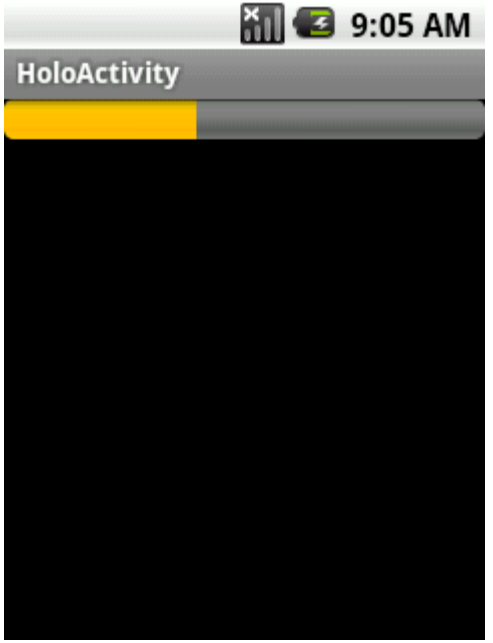
```
        isRunning = true;
        background.start();
    }

    public void onStop()
    {
        super.onStop();
        isRunning = false;
    }
}
```

Sending Messages to Handler

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <ProgressBar android:id="@+id/progress"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Sending Messages to Handler



Background Threads Caveats

- There is the risk that while the background thread executes the user will interact with the activity user interface and by doing so will invalidate the background thread. When something like that happens we should communicate the change to the the background thread.

Background Threads Caveats

- When a background thread is executed the possibility that in the meantime the activity will terminate exists. We should take that into consideration and pause (or kill) the background thread within the relevant callback method\s.

Background Threads Caveats

- Background threads might consume resources beyond expected and cause the user to lose his patience. We should take that into consideration and ensure we don't have redundant background threads that damage the user experience.

Background Threads Caveats

- During the execution of a background thread errors might occur. We should take that into consideration. In most cases the proper approach would be informing the user about the error.