App functionality beyond user interface-

Threads, Async task, Services

Creating multi-thread applications for Android application development is a challenging task for many Android developers. Single and multi-threading approaches are used to create complex Android enterprise mobile apps, as they help to streamline functional operation of the code. But sometimes it is necessary to update the UI from the background thread about the operations performed. If you ever tried to access an UI element from a background thread, you have already noticed that an exception is thrown. This article will explain how to notify activity with the information posted by another thread.

For creating multi-thread apps, Android by default does not allow the developer to modify the UI outside of the main thread. This problem is faced by many coders and if you still managed to do it you'd be breaking the second rule of the single-threaded model which is "*do not access the Android UI toolkit from outside the UI thread*", as stated here http://developer.android.com/guide/components/processes-and-threads.html.

Problem

While creating complex multi-thread functions in an enterprise android app, the information generated from these threads are not notified in the UI resulting in mismatch of app business logic and crashing of the application.

Solution

- Implement a Handler class, override method handleMessage() which will read messages from thread queue
- Next post message using sendMessage() method in worker thread

There are many situations when it is required to have a thread running in the background and send information to main Activity's UI thread. From the architectural level we can use two different approaches for notifying thread activity.

1. Use of Android AsyncTask class
2. Start a new thread

Using AsyncTask is very convenient as there might be situations when you really need to construct a worker thread by yourself. In such situation, you will need to send some information back to Activity thread. Keep in mind that Android doesn't allow other threads to modify any content of main UI thread as stated above. Instead you're required to wrap data into messages and send them through message queue.
You can implement this operation in two parts:

Part 1 – Add Handler

Add an instance of Handler class to your MapActivity instance.

```
public class MyMap extends MapActivity {
. . . .
public Handler _handler = new Handler() {
@Override
public void handleMessage(Message msg) {
Log.d(TAG, String.format("Handler.handleMessage(): msg=%s", msg));
// This is where main activity thread receives messages
// Put here your handling of incoming messages posted by other threads
super.handleMessage(msg);
}
};
. . . .
}
```

Part 2 – Post Message

In the worker thread post a message to activity main queue whenever you need Add handler class instance to your MapActivity instance.

```
/**
* Performs background job
*/
class MyThreadRunner implements Runnable {
// @Override
public void run() {
while (!Thread.currentThread().isInterrupted()) {
// Just dummy message -- real implementation will put some meaningful data in it
Message msg = Message.obtain();
msg.what = 999;
MyMap.this._handler.sendMessage(msg);
// Dummy code to simulate delay while working with remote server
try {
Thread.sleep(5000);
} catch (InterruptedException e) {
Thread.currentThread().interrupt();
}
}
```

}
}

AsyncTask

AsyncTask enables proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask is designed to be a helper class around Thread and Handler and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the java.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called Params, Progress and Result, and 4 steps, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.

public abstract class AsyncTask
extends Object

java.lang.Object
  ↳ android.os.AsyncTask<Params, Progress, Result>
AsyncTask must be subclassed to be used. The subclass will override at least one method (doInBackground(Params...)), and most often will override a second one (onPostExecute(Result).)

AsyncTask's generic types

The three types used by an asynchronous task are the following:

1. Params, the type of the parameters sent to the task upon execution.
2. Progress, the type of the progress units published during the background computation.
3. Result, the type of the result of the background computation.

The 4 steps

When an asynchronous task is executed, the task goes through 4 steps:

1. onPreExecute(), invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.
2. doInBackground(Params...), invoked on the background thread immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use publishProgress(Progress...) to publish one or more units of progress. These values are published on the UI thread, in the onProgressUpdate(Progress...) step.
3. onProgressUpdate(Progress...), invoked on the UI thread after a call to publishProgress(Progress...). The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.
4. onPostExecute(Result), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Threading rules

There are a few threading rules that must be followed for this class to work properly:

- The AsyncTask class must be loaded on the UI thread. This is done automatically as of JELLY_BEAN.
- The task instance must be created on the UI thread.
- execute(Params...) must be invoked on the UI thread.
- Do not call onPreExecute(), onPostExecute(Result), doInBackground(Params...), onProgressUpdate(Progress...) manually.
- The task can be executed only once (an exception will be thrown if a second execution is attempted.)