Battle-Tested Patterns in Android Concurrency Part 1

Doug Stevenson
Friday, July 31, 2015
8:30 AM

Sample code:
https://github.com/AnDevDoug/concurrency

Why Threading and Concurrency?

- Smooth, responsive UI while performing background work
- Speed things up using multiple cores
- Improve your engineering skills

Most Important Concerns

Keep I/O and heavy CPU work off the main thread

Why: Avoid janky UI, ANRs

Includes: File access, database work, network access
(use strict mode dev preference and `android.os.StrictMode`)

All UI updates (changes to the View hierarchy) must be on the main thread

Why: Android will enforce it (your app will crash)

Most Important Concerns

Don’t leak Activity references

Why: Risk of running out of memory

Design for thread safety up front

Why: Or your users will discover the edge cases and give you bad ratings

One Concurrency Solution: Plain Java Threads

Plain Java Threads

What they do:

- Whatever you tell the threads to do

When to use them:

- You need full control over threading behavior
- You fully understand the concurrency behavior of the entirety of your app

What they DO NOT do:

- Handle activity lifecycle and configuration changes
- Facilitate UI updates

Plain Java Thread Example

```
private TextView tv;

protected void onCreate(Bundle) {
    ...
    tv = (TextView) findViewById(...);

    new Thread() {
        public void run() {
            // load the result string from some blocking data source
            final String result = ???;
            runOnUiThread(new Runnable() {
                public void run() {
                    tv.setText(result);
                }
            }
        }
    }.start();
}
```

Plain Java Thread Example

What could go wrong here?

- Possibly leaking Activity reference
  (non-static inner classes contain an implicit hard reference to any outer classes)
- tv instance is no longer visible to the user after the activity finishes

Plan Java Thread Demo

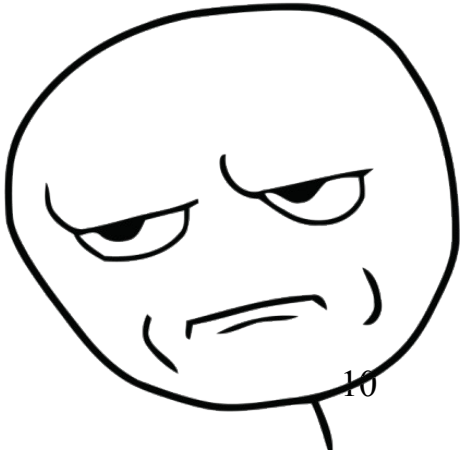
See ActivityBasicThread.java

Plain Java Thread Example

Anti-pattern Fix #1:

- Setting the activity's screenOrientation attribute in the manifest

```
<activity android:screenOrientation="portrait" />
```

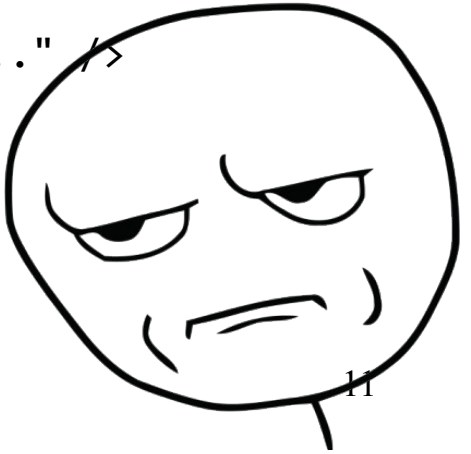


Plain Java Thread Example

Anti-pattern Fix #2:

- Setting the activity's configChanges attribute in the manifest

```
<activity android:configChanges="orientation|keyboardHidden" />
<activity android:configChanges="orientation|keyboardHidden|...|..." />
```



Tips for Using Java Threads

#1 Have a strategy for dealing with configuration changes.

Handle Activity start/stop

- Interrupt/quit the thread and save work in onDestroy (or onStop)
- Resume work in onCreate (or onStart) of the new activity.

Rather maintain the running thread?

- Problematic; don't do this (but stay tuned!)

Tips for Using Java Threads

#2 Minimize the chance of uninterruptible work

- Fully close/abort socket transactions to stop connects and reads
- Check for thread interruption in CPU-bound loops

Tips for Using Java Threads

#3 Prevent Activity leaks

- Force a decoupling of Activity/View instances with Thread instances
- If needed, find a way to do UI updates

Summary: Using Java Threads

Avoid managing threads directly in your activities unless you absolutely know what you're doing!

Another Solution: Android's AsyncTask

Android's AsyncTask

What it does:

- Provides a mechanism to put one or more uniform units of work in a separate thread
- Results of work units are individually published to the main thread

When to use it:

- You have several small, quick things to do in an activity that makes changes to the UI

For example:

- Decoding bitmaps
- Repeated database queries

Android's AsyncTask

What it DOES NOT do:

- Does NOT handle activity lifecycle and configuration changes
- Does NOT behave consistently between different Android versions
  - < 1.6, all AsyncTasks shared a single thread
  - 1.6 <=> 2.3, AsyncTasks shared a thread pool of 5 threads max
  - >= 3.0, back to single shared thread
  - >= 3.0, API to choose an Executor to run AsyncTasks on

Android's AsyncTask

NOT recommended for:

- Long running operations
- Network I/O

AsyncTask Usage

AsyncTask must be subclassed with generics:

```
public class YourAsyncTask extends AsyncTask<Params, Progress, Result>
```

- Params is the work unit input data type
- Progress is the work unit progress data type
- Result is the overall result of the task
- Any type may be Void if unused

AsyncTask Code Structure

```
class SampleAsyncTask extends AsyncTask<Params, Progress, Result> {
    @Override
    protected void onPreExecute() {
        // OPTIONAL: Called on the main thread for init
    }
    @Override
    protected Result doInBackground(final Params... params) {
        // REQUIRED: Iterate and process params on background thread.
        // Call publishProgress(Progress...) to send results to main thread.
        // Return a Result.
    }
    @Override
    protected void onProgressUpdate(final Progress... values) {
        // OPTIONAL: Called on main thread in response to publishProgress()
    }
    @Override
    protected void onPostExecute(final Result result) {
        // OPTIONAL: Called on the main thread after all background work is done
    }
}
```

AsyncTask Usage

Pass one or more units of work to AsyncTask by calling:

```
public final AsyncTask execute(Params... params)
```

Cancel an AsyncTask using:

```
public final boolean cancel(boolean mayInterruptIfRunning)
```

Check to see if canceled during doInBackground():

```
public final boolean isCancelled()
```

AsyncTask Summary

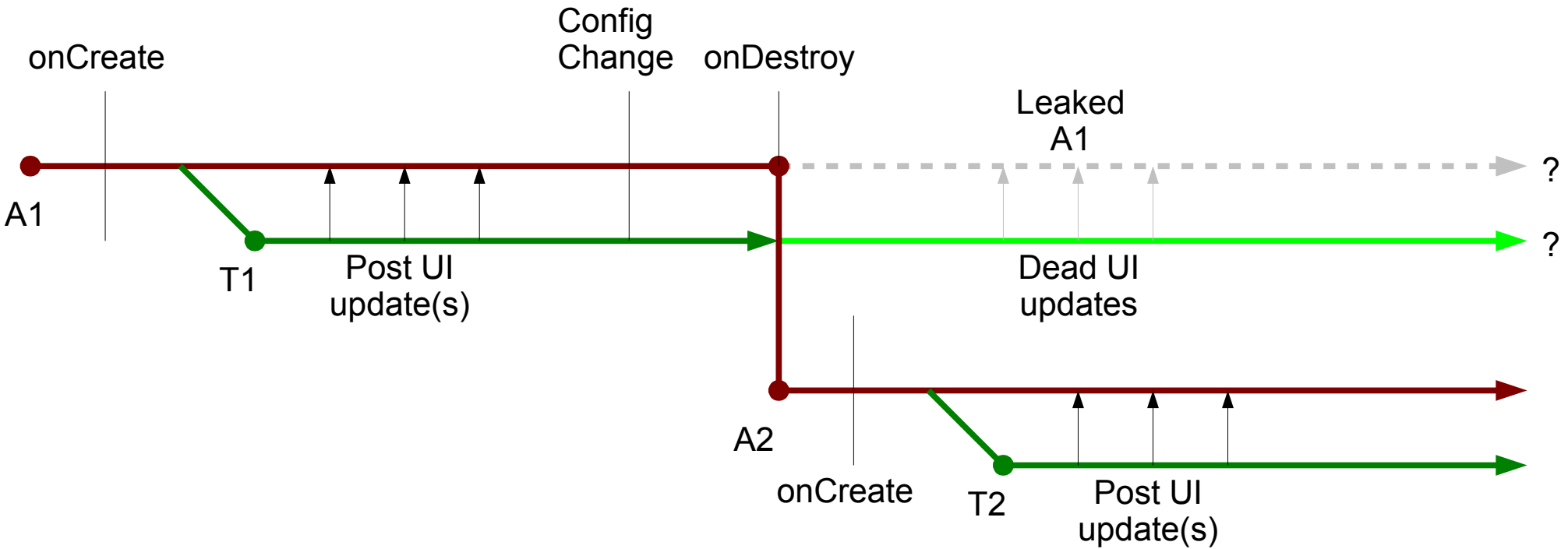
- Better than managing Java Threads
- Helps with putting incremental results on the main thread
- Inconsistent behavior on different API levels
- Still can leak an Activity if not careful

AsyncTask Demo

See ActivityBasicAsyncTask.java

Quick Detour:
Activity Leaks

Visualizing Activity Leaks



Possible Causes of Activity Leaks

- T1 doesn't end quickly after onDestroy
  - Forgotten? Uninterruptible? Blocked? Busy loop?
- **AND**: T1 prevents A1 to be garbage collected
  - Strong reference to A1 (or one of its views)
  - Indirect strong ref (inner classes see outer class instance)

Avoiding Activity Leaks

- Make your Thread/AsyncTask subclass static (if inside the Activity) or a standalone class
- In the constructor, pass in the Activity object and use a WeakReference to hold it.
- Check the Activity WeakReference contents for null on each access
- Remember to end thread execution no later than onDestroy

Services

Services

Quick overview:

- Android app component
- A Service is a Context
- Lifecycle independent of Activities
- Don't restart with configuration changes
- Instances must provide their own threading behavior
- Can be a “started” service or a “bound” service, or both
  - Only dealing with started services here

Started Services

Use a started service when your background work:

- Must continue beyond the Activity that initiated it
- May be started at any time and may run indefinitely

For example:

- Large background uploads, downloads, data refresh, sync
- Lengthy computation
- Background media playback
- Other background operations that the user should be aware of

Started Services

Be careful:

- Manage the lifecycle of the service
- Manage threading directly or use IntentService behavior
- Figure out how to publish data to other parts of the app

IntentService

- Single thread per service
- All work queued and serialized on that thread
- Service is “started” when work is active or pending
- No more work? Worker thread ends and service stops

IntentService Usage

- Subclass IntentService
- Add the Service to AndroidManifest.xml
- Override onHandleIntent(Intent)
- Logic in onHandleIntent parameterized by the contents of the intent (action, extras)
- Clients initiate work using context.startService(Intent)
  - Intent instance uses the class of the IntentService subclass

IntentService Example: Client

```
Intent intent = new Intent(context, YourIntentService.class);
intent.setAction("ACTION");
intent.putExtra("repeat", 5);
startService(intent);
```

IntentService Example: Service

```
public class YourIntentService extends IntentService {

   public MyIntentService() {
      super("YourIntentService");
   }

   @Override
   protected void onHandleIntent(Intent intent) {
      // Called on background thread, take action on intent

      String action = intent.getAction(); // could use action as switch
      if ("ACTION".equals(action)) {
         int repeat = intent.getIntExtra("repeat", 5);
         // Do stuff
      }

   }

}
```

IntentService Demo

See BasicIntentService.java and ActivityBasicIntentService.java

Looper, Handler, HandlerThread

Looper

- Implements a message loop/queue/pump on a Thread
- One Looper → One Thread
- Looper logic:
  1. Wait for work
  2. Execute work
  3. Goto 1

Handler

- Schedules work on a Thread with a Looper
- Messages may be scheduled
  - A message is a data payload
  - `sendMessage(), sendMessageAtTime(), sendMessageDelayed()`
  - Handler should provide implementation for message actions
- Runnables may be scheduled
  - `post(), postAtTime(), postDelayed()`
- One Handler → One Looper → One Thread
- One Thread → One Looper → Multiple Handlers

HandlerThread

- Convenience class for:
  - Starting a new Thread
  - Creating a Looper on it
- Once started, ready for new Handlers to give it work

```
HandlerThread handlerThread = new HandlerThread("Name", priority);
handlerThread.start();
Looper looper = handlerThread.getLooper();
Handler handler = new YourHandler(looper);
// Now post runnables and messages to handler for exec on thread...
looper.quit();
```

Part 2 at 11am

Main topic: Loaders & More

Sample code:
https://github.com/AnDevDoug/concurrency