Battle-Tested Patterns in Android Concurrency

Doug Stevenson
Friday, December 7, 2012 2:00pm

Slides gzip: http://goo.gl/53eVv
Code Samples: http://goo.gl/pL1tK

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 ANR!

Includes: File access, database work, network access
(use strict mode to find blocking calls)

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 concurrency correctness up front

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

Possible Solution: Direct 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

Direct 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();
}
```

Direct Java Thread Example

What could go wrong here?

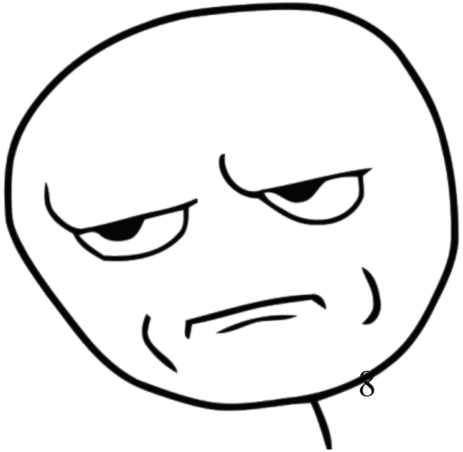
- Leaking Activity reference
  (non-static inner classes contain an implicit hard reference to any outer classes)
- tv instance is not longer visible to the user after configuration change

Direct Java Thread Example

Anti-pattern Fix #1:

- Setting the activity's screenOrientation attribute in the manifest

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

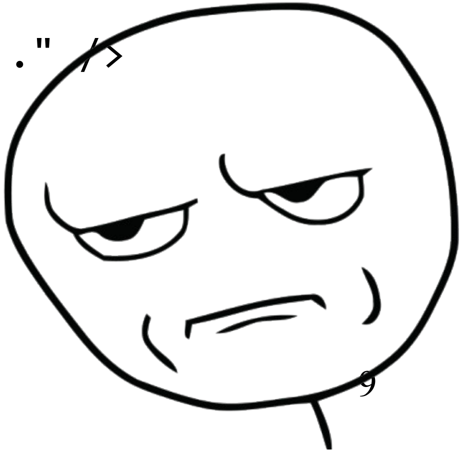


Direct Java Thread Example

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?

- Don't do this (but stay tuned!)

Tips for Using Java Threads

#2 Minimize the chance of uninterruptible work

- Use connect and read timeouts with socket I/O and HTTP libs
- 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

Do not use threads directly in your activities unless you absolutely know what you’re doing!

Possible Solution: AsyncTask

What it does:

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

When to use it:

- You have several (small) uniform things to do in an activity that makes changes to the UI

For example:

- Decoding bitmaps
- Repeated database queries

Possible Solution: 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

Possible Solution: AsyncTask

NOT recommended for:

- Long running operations
- Network I/O

AsyncTask Usage

AsyncTask must be subclassed specifying its generic types:

```
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

```
public 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 Result
    }
    @Override
    protected void onProgressUpdate(final Progress... values) {
        // Called on main thread for each call 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 AsyncTask execute(Params... params)
```

Cancel an AsyncTask using:

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

Check to see if canceled in 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

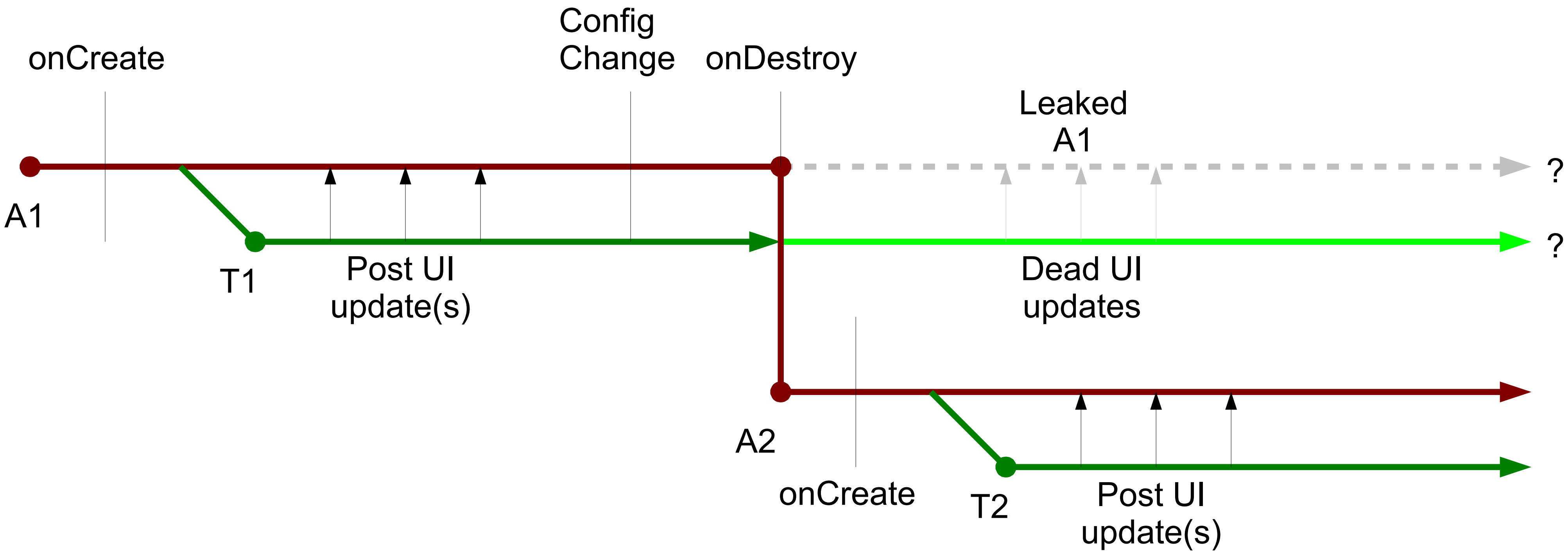
AsyncTask Demo

See ActivityBasicAsyncTask.java

AsyncTask: Avoiding Activity Leaks

- Make the 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 cancel it no later than onDestroy

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
  - T1 holds a direct *strong ref* to A1, e.g.:

    T1: `A1.getResources().getString(...);`
  - **OR**: T1 is an *inner member class* of A1 (implicit strong ref to A1)
  - **OR**: T1 holds an *indirect strong ref* to A1, e.g.:

    A1: Manages a Handler that performs UI updates

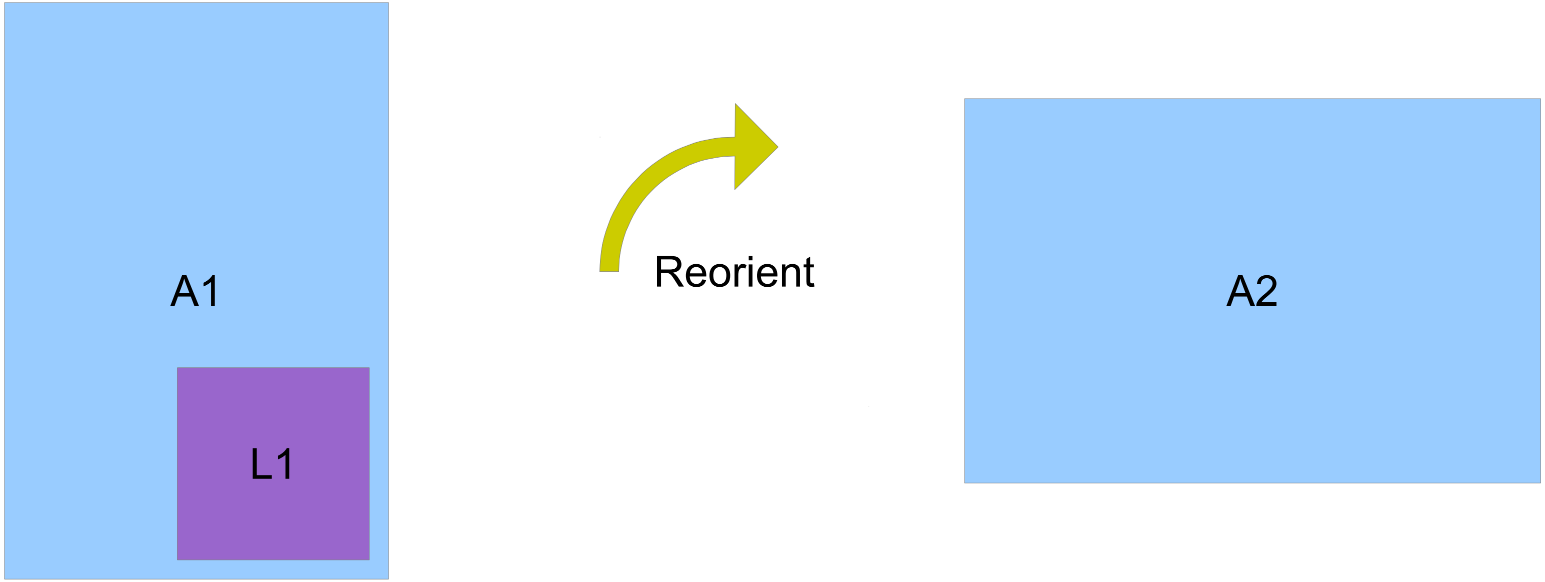
    T1: `Handler.post(message)`

Possible Solution: Loaders

What they do:

- Put a single unit of work on separate thread
- Delivers the result of the work on the main thread
- Continue pending work during a configuration change
- Remember the result between activity configuration changes
- Can monitor a source of data for change, notifying the Activity on change (e.g. CursorLoader)

Loader Illustrated



Possible Solution: Loaders

Use them when:

- You have discrete data to fetch or compute in a single activity that will update the UI
- You have work to do that must survive the Activity lifecycle after a configuration change

For example:

- Load data (from a database, file, network) for display

Loader Notes

Loaders first available in API level 11 (Honeycomb):

- android.content.Loader
- android.app.LoaderManager
- subclass android.app.Activity

Loaders available to API level 4 via Android Compatibility library:

- android.support.v4.content.Loader
- android.support.v4.app.LoaderManager
- subclass android.support.v4.app.FragmentActivity

Loader Usage

Two parts to using a loader:

1. Loader class
    - Subclass of Loader android.support.v4.content.Loader
    - Performs background work in another thread
    - Instances managed by LoaderManager
2. LoaderCallbacks class
    - Impl android.support.v4.app.LoaderManager.LoaderCallbacks
    - Creates the Loader instance to use
    - Receives the Loader's results, updates the Activity UI

Loader Usage: Loader Class

```
public class YourLoader extends SomeBaseLoader<Result> {

    private final int arg;
    public YourLoader(final Context context, final int arg) {
        super(context);
        this.arg = arg;
    }

    @Override
    protected Result loadInBackground() {
        Result result;
        // use arg to load Result in the background
        return result;
    }

}
```

Tips for Loaders

- Should never contain Activity instances
- May never be non-static inner class in an activity (enforced)
- The Loader decides how to background blocking work

Loader Usage: LoaderCallbacks Class

```
public class YourLoaderCallbacks implements LoaderCallbacks<Result> {

    @Override
    public Loader<Result> onCreateLoader(final int loader_id, final Bundle args) {
        // Create the Loader instance; pass stuff from args into it if necessary
        return new YourLoader(context, args.getInt("key"));
    }

    @Override
    public void onLoadFinished(final Loader<Result> loader, final Result data) {
        // Do something with the loaded result in the UI
    }

    @Override
    public void onLoaderReset(final Loader<Result> loader) {
        // typically empty
    }

}
```

Tips for LoaderCallbacks

- Typically implemented as Activity inner classes
- May contain/use Activity instances without leaking
- But don't pass Activity instances through to the Loader!
- Typically take parameters from the args Bundle
- (but you don't have to pass params that way)

LoaderManager

Manages instances of Loaders between Activity instances.

For one-time loads, typically done during onCreate():

```
LoaderManager lm = getSupportLoaderManager();
LoaderCallbacks<Result> callbacks = new YourLoaderCallbacks();
Loader<Result> loader = lm.initLoader(
    LOADER_ID,
    (Bundle) null,
    callbacks
);
```

LoaderManager.initLoader

Two circumstances to remember when you call initLoader()

1. If the Loader with the given id IS NOT already created:
   - The given LoaderCallbacks is asked to create a new one
   - The new loader starts its loadInBackground()
2. If the Loader with the given id IS already created:
   - The given LoaderCallbacks is associated with the existing Loader
   - If the load is already complete, callbacks will be notified next cycle

Other LoaderManager Methods

```
Loader<T> getLoader(int id)
```

Returns the Loader with the given id or null if not running.

```
void destroyLoader(int id)
```

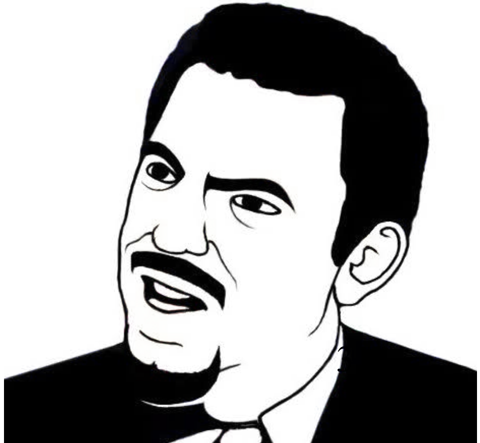
Stops the Loader with the given id.
If already finished work, calls LoaderCallbacks.onLoaderReset.

```
Loader<T> restartLoader
    (int id, Bundle args, LoaderCallbacks<T> data)
```

If Loader not already running, works like initLoader.
If Loader already running, it will be destroyed first.

About Android’s AsyncTaskLoader

- AsyncTaskLoader is a Loader implementation by Google
- Implemented on top of AsyncTask
- Inherits all of AsyncTask’s idiosyncrasies
- Doesn’t always work exactly as a Loader should: http://code.google.com/p/android/issues/detail?id=14944



ExecutorServiceLoader: A better loader

- Default operation puts all work on a singleton thread
- Or you can give it an ExecutorService to handle threading
- If you give it an ExecutorService, make it also a global singleton
  - DO NOT create a new ExecutorService in onCreate()
- Requires your results to be a ResultOrException<T, E>
  - Data container for a generic result type or an Exception subclass
  - Handy because Loaders can't “throw”, but can generate errors
  - Must check if result or exception exists before using either

Basic Loader Demo

See ActivityBasicLoader.java

Loader as a Non-static Inner Class

Loader helps defend against accidental Activity leaks:

- All Loader classes are required NOT to be a non-static inner class.
- Non-static inner class loaders will make your app will crash:

```
java.lang.IllegalArgumentException: Object returned from onCreateLoader
must not be a non-static inner member class
```

- Nothing stopping you from injecting an Activity into a custom loader.

Non-Static Loader Demo

See ActivityInvalidNonStaticLoader.java

A Tricky Situation with a Loader

1. You have a Button that:
   - Kicks off a Loader (NOT in onCreate())
   - Updates UI to disable the button and show a wait spinner
2. Configuration change → new Activity
3. New Activity needs to reattach the Loader and disable the button

Problems:

- You can't blindly call initLoader() in onCreate()
- getLoader() won't tell you if the loader is in progress or finished

Stateful Loader Part 1

Create a Loader subclass with a method getState() that returns an enum for load state (e.g. Loading, Loaded).

```
public class StatefulLoader extends BaseLoader<Result> {

    private volatile State state;

    public static enum State { Loading, Loaded; }

    public State getState() { return state; }

    protected Result loadInBackground() {

        state = State.Loading;
        // Do your loading here
        state = State.Loaded;

        return result;

    }
}
```

Stateful Loader Part 2

Then in onCreate():

```
initViews();

LoaderManager lm = getSupportLoaderManager();
Loader<Result> loader = lm.getLoader(LOADER_ID);
StatefulLoader statefulLoader = (StatefulLoader) loader;

if (statefulLoader != null) {

    lm.initLoader(LOADER_ID, null, new YourLoaderCallbacks());

    switch (statefulLoader.getState()) {
    case Loading:
        updateUiLoading();
        break;
    case Loaded:
        break;
    }

}
```

Another Trick for Saving Loader State

How to keep track of multiple potential loaders?

1. Remember all Loader ids that have been init'd
2. In onSaveInstanceState(), save all init'd loader ids in the state bundle
3. In onCreate():
    a) Get list of saved loader ids from the Bundle arg
    b) Check their state, update UI
    c) Init each each loader id again

Tracking Loader Work Progress

If you have a Loader that should track incremental progress:

- Start with Stateful Loader pattern
- Use LocalBroadcastManager as a data exchange mechanism
- In the Loader, broadcast progress updates
- Implement a BroadcastReceiver to handle progress updates
- Register the BroadcastReceiver in onCreate() / Unregister in onDestroy()

Tracking Loader Demo

See ActivityProgressLoader.java

CursorLoader: Loading from ContentProvider

To use Android's CursorLoader, you need a Uri for a ContentProvider:

- From an Android system component (Calendar, Contacts, Media)
- From another app
- One you created for yourself

http://developer.android.com/guide/topics/providers/content-providers.html

CursorLoader Usage

1. Create a LoaderCallback that implements LoaderCallbacks<Cursor>
2. In onCreateLoader(), create and return a CursorLoader with the ContentProvider query
3. In onLoadFinished(), make use of the Cursor

CursorLoader Callbacks Example

```
public class YourLoaderCallbacks implements LoaderCallbacks<Cursor> {

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        return new CursorLoader(activity, content_uri, ...);
    }

    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        listView.setAdapter(new YourCursorAdapter(activity, data, 0));
    }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
    }

}
```

CursorLoader Demo

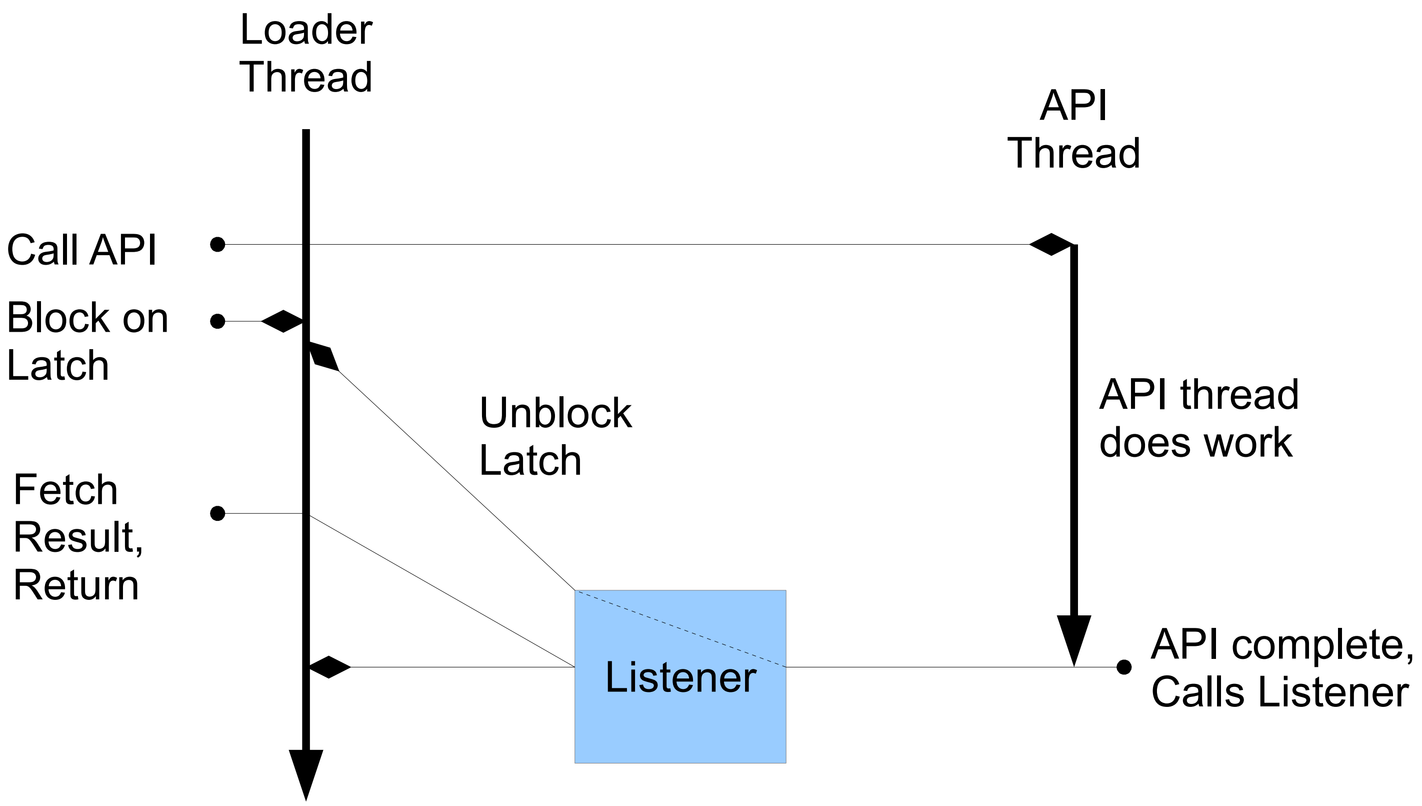
See ActivityMusicCursorLoader.java

Loaders and Asynchronous APIs

Using a fully asynchronous API?
(methods return immediately, work is on another thread, calls a listener on completion)

1. Call the API in loadInBackground()
2. Use a CountDownLatch to block the Loader thread and wait for a result
3. Have the listener signal the Latch and store the API result
4. After the Latch unblocks the main thread, fetch the result and return it

Loaders and Asynchronous APIs



When Not to Use a Loader

- Lots of little downloads (e.g. thumbnails)
- The load needs to continue after the activity is done

Loader Summary

- Loader is a useful and underutilized tool for Android development.
- Some boilerplate overhead in coding, but addresses the most common problems with background tasks in Activities

Services

Quick overview:

- Android app component
- A Service is its own 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? Thread goes away 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)
- Initiate work using context.startService(Intent)
  - Intent instance uses the class of the IntentService subclass

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 Example: Client

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

IntentService Demo

See IntentServiceBasic.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 message
    2. Execute message
    3. Goto 1

Looper, Handler, HandlerThread

Handler

- Schedules and executes work on a Thread with a Looper
- Messages may be scheduled
  - sendMessage(), sendMessageAtTime(), sendMessageDelayed()
- Runnables may be scheduled
  - post(), postAtTime(), postDelayed()
- One Handler → One Looper → One Thread
- One Thread → Multiple Handlers

Looper, Handler, HandlerThread

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();
```

Optimizing Intermittent Network I/O

- e.g. High throughput remote API calls with small payloads
- Limit to one or two threads to prevent saturating a slow connection
- Maybe increase threads if connection speed is high

Optimizing Sustained Network I/O

- e.g. Downloading large files and images
- Limit to just one thread to prevent saturating a slow connection

Optimizing File I/O

- e.g. Simple database queries that can touch many rows
- e.g. Access to external storage
- Limit to just one thread (per storage device) to prevent I/O thrashing

Optimizing Heavy CPU Work

- e.g. Decoding many/large bitmaps (or any media)
- e.g. Performing complex database operations (could be I/O intense as well)
- Limit to number of CPU cores minus one
  - `Runtime.getRuntime().availableProcessors();`

Other concurrency tips

- Android has all the same thread tools and issues as Java 5.
  - java.util.concurrent.* (Semaphore, BlockingQueue, etc)
- Devices may not power up all their CPU's cores immediately
- Avoid polling loops at any cost
- Smartly use android.os.Process.setThreadPriority(prio)
  - android.os.Process.THREAD_PRIORITY_BACKGROUND
  - android.os.Process.THREAD_PRIORITY_AUDIO
  - (Do NOT use java.lang.Thread.setPriority())

Other concurrency tips

- Consider Renderscript Computation (API 11+) to offload heavy math to the GPU
  - http://developer.android.com/guide/topics/renderscript/compute.html

Feedback?

Please use the EventBoard app!