Solving an Android Threading Problem

by
Eric M. Burke, Principal Software Engineer
Object Computing, Inc. (OCI)

Introduction

By now, you probably know what Google Android is: an open source operating system, virtual machine, and SDK for mobile devices. In 2008, T-Mobile released the first Android phone, the G1. 2009 will bring many different phones from a variety of carriers.

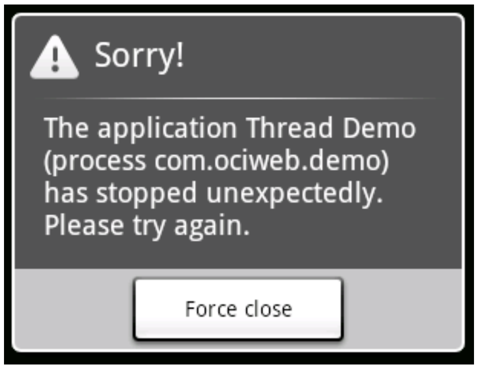
Android presents an exciting opportunity for programmers. Millions of people will purchase Android phones in 2009, each including a link to the Android Market. For a nominal $25 registration fee, any programmer can distribute free Android applications on the Market. Beginning in January, you'll be able to sell commercial applications, as well.

If you are new to Android development, start with Google's Notepad Tutorial. Rather than repeat that material, this article looks at a particular threading problem in more detail. If you are completely new to Android, you may want to work through the Notepad tutorial before proceeding.

Threading Rules

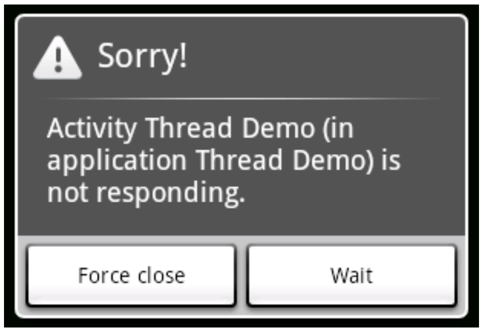
Like other GUI toolkits, the Android user interface is single-threaded. To avoid locking up the GUI, long running operations *must* run in background threads. This should sound familiar to Swing programmers, although Android differs in two notable ways:

- Android *fails fast* when background threads update GUI components. Rather than silently ignoring this kind of threading bug, Android throws `CalledFromWrongThreadException` and immediately terminates the activity.


Sorry!

The application Thread Demo (process com.ociweb.demo) has stopped unexpectedly. Please try again.

- If a long-running process locks up the UI, Android intervenes and displays this dialog to the user:

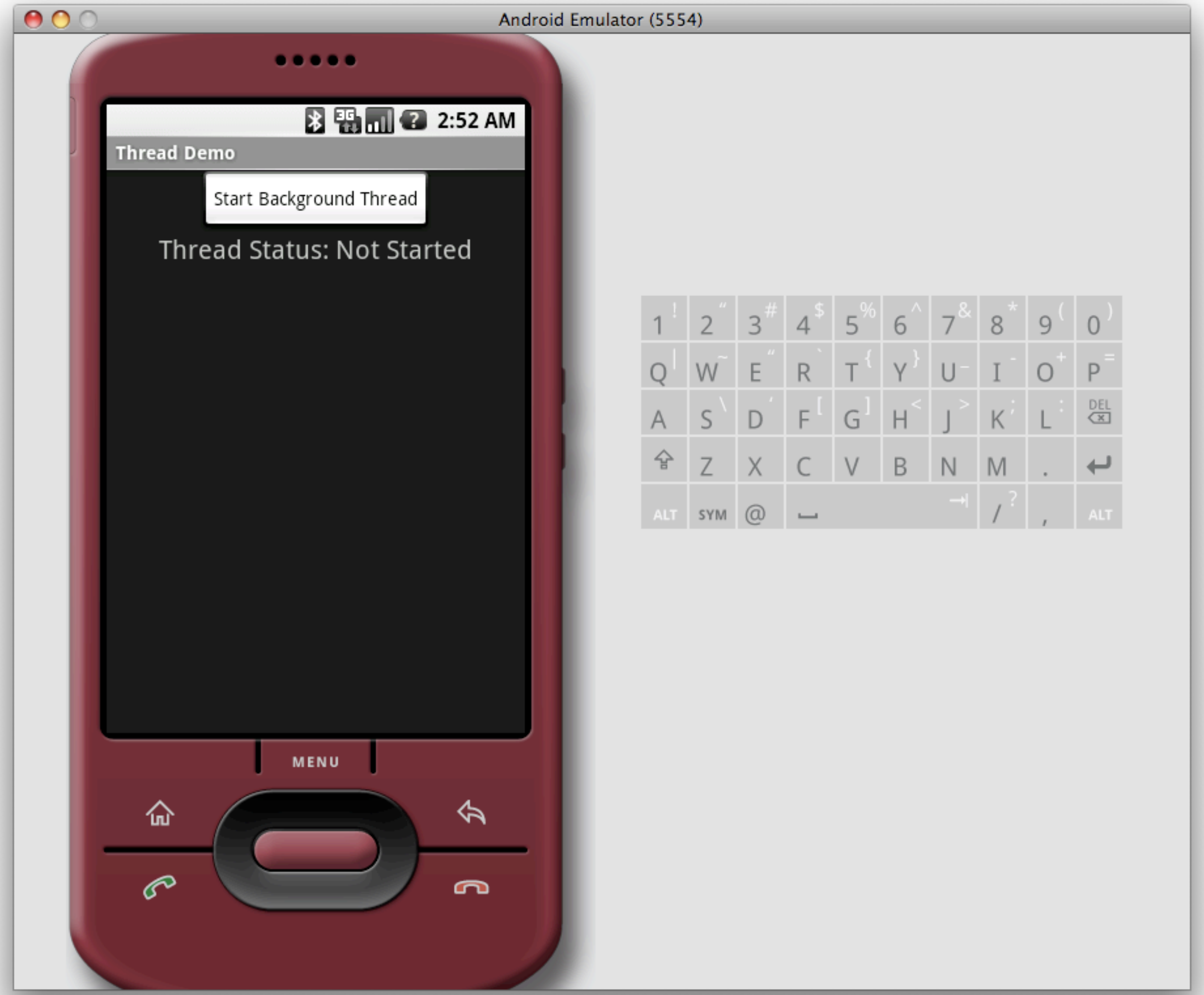

Sorry!

Activity Thread Demo (in application Thread Demo) is not responding.

These are welcome improvements because they encourage correct code and help programmers locate bugs early in the development process. They also prevent poorly written applications from locking up your entire phone.

The Example

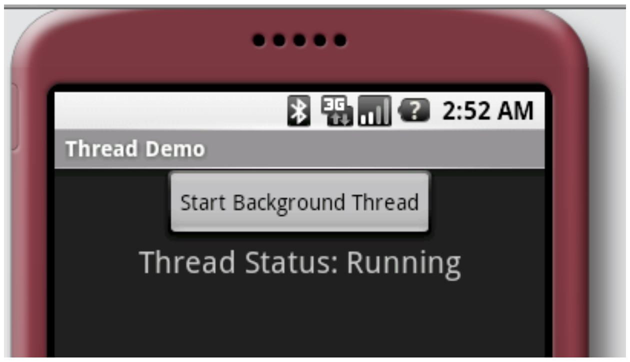
Our sample application looks like this when running in the Android emulator. (The emulator is included in the free SDK.)



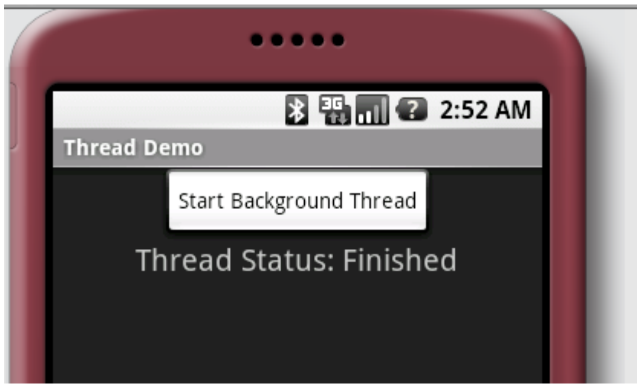
When you click *Start Background Thread*, a few things happen:

1. The button is disabled
2. The status label changes to *Running*
3. A background thread simulates a long operation

While running, the UI looks like this:



When the thread completes, the button becomes enabled again and the status label shows *Finished*.



Now let's review the source code.

XML Files

Our application contains three XML files, larger applications will have many more. Although Android utilizes XML during application development, all of these XML files are compiled into a highly efficient binary form.

AndroidManifest.xml

Every Android application has a manifest in its root folder. The Android Eclipse plugin generates ours, specifying `HomeActivity` as the application entry point.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.ociweb.demo"
          android:versionCode="1"
          android:versionName="1.0.0">
  <application android:icon="@drawable/icon"
               android:label="@string/app_name">
    <activity android:name=".HomeActivity"
              android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Although this XML is verbose, editing is easy thanks to the graphical editors in the Android Eclipse plugin.

strings.xml

The next XML file, *strings.xml*, defines labels for the entire application. As you can see, the `app_name` used in *AndroidManifest.xml* is defined here in *strings.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Thread Demo</string>
  <string name="start_background_thread">Start Background Thread</string>
  <string name="thread_running">Running</string>
  <string name="thread_finished">Finished</string>
  <string name="thread_status">Thread Status:</string>
  <string name="thread_not_started">Not Started</string>
</resources>
```

As you add new values to this (and other) XML files, the Eclipse plugin automatically generates a file named `R.java` with constants for each string. The Android SDK also includes command line tools that generate `R.java` if you don't use Eclipse.

home.xml

This is the GUI layout file for the home screen shown earlier. Android lets you define screen layout in XML files or programmatically, although XML is generally preferred. Like other XML files, layout files are compiled to a more efficient form before they are ever installed on a phone.

```
<?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"
    >

  <Button android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:id="@+id/start_background_thread_btn"
          android:text="@string/start_background_thread"
          android:layout_gravity="center"/>

  <LinearLayout android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:orientation="horizontal"
         android:layout_gravity="center">

    <TextView android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:text="@string/thread_status"
         android:textSize="20dp"
         android:paddingRight="5dp"/>

    <TextView android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:id="@+id/thread_status_label"
         android:text="@string/thread_not_started"
         android:freezesText="true"
         android:textSize="20dp"/>
  </LinearLayout>
</LinearLayout>
```

Android includes a variety of layout classes, such as `LinearLayout`, `FrameLayout`, `RelativeLayout`, and more. Like Swing layout managers, these help your UI adjust to varying screen resolutions with minimum hardcoding.

Rather than include labels in the layout XML files, we choose to reference values in *strings.xml* using this syntax: `@string/thread_status`.

You may also notice these identifiers: `@+id/thread_status_label`. The `@+id` tells the Android tools to generate a constant in *R.java*. Your application code always uses *R.java* constants to ensure a degree of compile-time safety.

Finally notice the `android:freezesText="true"` attribute on the status label. This ensures the label remembers its value when you change screen orientation.

Java Source

R.java

The Android SDK includes a command-line tool named *aapt* that generates *R.java* from the XML files. If you use Eclipse, the plugin instantly updates *R.java* whenever you save changes to one of the XML files. This is very useful when you rename constants because you get compile-time errors until you also update the code.

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package com.ociweb.demo;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int start_background_thread_btn=0x7f050000;
        public static final int thread_status_label=0x7f050001;
    }
    public static final class layout {
        public static final int home=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int start_background_thread=0x7f040001;
        public static final int thread_finished=0x7f040003;
        public static final int thread_not_started=0x7f040005;
        public static final int thread_running=0x7f040002;
        public static final int thread_status=0x7f040004;
    }
}
```

Finally, using integer constants is more efficient than looking up strings from XML.

HomeActivity.java

Other than the *home.xml* layout file, most of what we've seen so far is either generated by command line tools or edited via graphical tools in Eclipse. The actual home screen, however, is Java source code. In Android, an *Activity* is something the user can do. In most cases, Activities are screens in the user interface.

`HomeActivity` extends Android's `Activity` base class. The Android APIs make heavy use of the Template Method pattern, in which base classes like `Activity` define numerous methods you must override.

```
public class HomeActivity extends Activity implements OnClickListener {

  private Button mStartButton;
  private TextView mStatusLabel;

  private static final String ENABLED_KEY = "com.ociweb.buttonEnabled";

  // background threads use this Handler to post messages to
  // the main application thread
  private final Handler mHandler = new Handler();

  // post this to the Handler when the background thread completes
  private final Runnable mCompleteRunnable = new Runnable() {
    public void run() {
      onThreadCompleted();
    }
  };
```

We'll see how to initialize `Button` and `TextView` shortly. The `ENABLED_KEY` let's us remember if the button is enabled when the user changes the screen orientation. On the G1 phone, the screen rotates from portrait to landscape mode when the user slides out the keyboard. This key comes into play in the `onCreate(...)` and `onSaveInstanceState(...)` methods.

The `Handler` shown above allows background threads to send messages or `Runnable` objects back to the main application thread. This is the same concept as `SwingUtilities.invokeLater(Runnable r)` and `SwingUtilities.invokeAndWait(Runnable r)`. When the background thread completes, it passes the `mCompleteRunnable` to the `Handler` for execution on the main thread.

Next we see the `onCreate(...)` method.

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.home);

    mStartButton = (Button) findViewById(R.id.start_background_thread_btn);
    mStatusLabel = (TextView) findViewById(R.id.thread_status_label);

    mStartButton.setOnClickListener(this);


    if (savedInstanceState != null) {
      if (savedInstanceState.containsKey(ENABLED_KEY)) {
        mStartButton.setEnabled(savedInstanceState.getBoolean(ENABLED_KEY));
      }
    }
  }
```

`onCreate(...)` is one of several Activity Lifecycle methods, called at appropriate times as Activites come and go. Android always calls `onCreate(...)`, so this is a good place to locate GUI components and register event listeners. The code also shows how we use the generated *R.java* constants.

This is also where we re-establish the enabled flag on `mStartButton`, using the `ENABLED_KEY` defined earlier. Android uses `savedInstanceState` for short-term data storage, such as when activities are temporarily paused or when screen rotation happens.

The enabled flag is saved here in the `onSaveInstanceState(...)` method:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);
  outState.putBoolean(ENABLED_KEY, mStartButton.isEnabled());
}
```

Next, our Activity implements `onClick(View v)` from the `OnClickListener` interface. This is how we react to button clicks.

```
public void onClick(View v) {
  if (v == mStartButton) {

    mStartButton.setEnabled(false);
    mStatusLabel.setText(R.string.thread_running);
    // show a brief popup alert
    Toast.makeText(this, R.string.thread_running, Toast.LENGTH_SHORT).show();

    Thread t = new Thread() {
      public void run() {
        // perform expensive tasks in a background thread
        expensiveOperation();

        // let the UI know the task is complete
        mHandler.post(mCompleteRunnable);
      }
    };
    t.start();
  }
}
```

On a phone like the G1, users click buttons by tapping on the screen or hitting a physical button on the phone. Our code changes the status label, disables the button, and launches a background thread. Using this thread avoids locking up the main application thread.

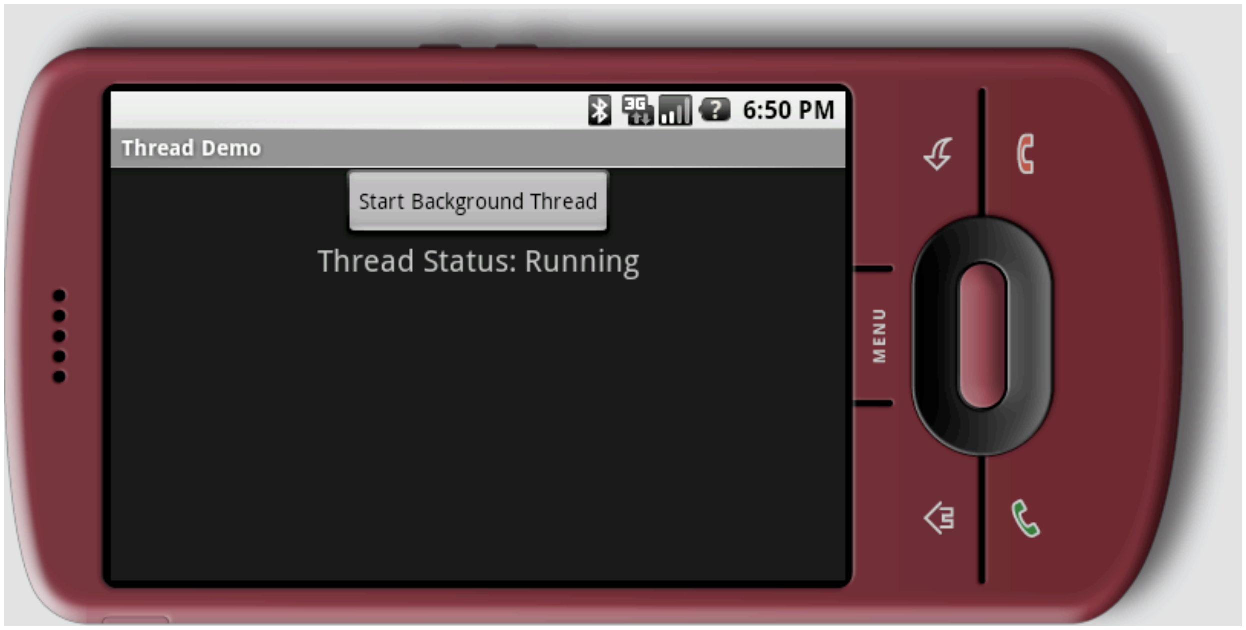
Once the thread completes, it posts the `mCompleteRunnable` to the `mHandler`. Our `Runnable`, in turn, calls the `onThreadComplete()` method, shown next.

```
 /**
   * Call this method on the main application thread once the background thread
   * completes.
   */
  private void onThreadCompleted() {
    mStartButton.setEnabled(true);
    mStatusLabel.setText(R.string.thread_finished);
    Toast.makeText(this, R.string.thread_finished, Toast.LENGTH_SHORT).show();
  }

  /**
   * This method runs in a background thread. In a real app, it would do
   * something useful, such as getting data from a web service.
   */
  private void expensiveOperation() {
    try {
      TimeUnit.SECONDS.sleep(4);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

Those calls to `Toast.makeText(...)` trigger a brief popup alert. When the thread is complete, we enable the button again and change the status label back to "Finished".

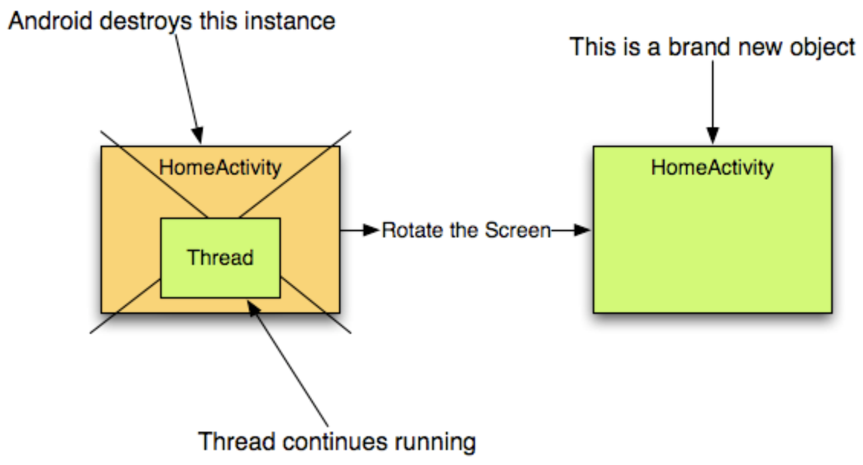
The Threading Bug

The example handles threads just like most of the examples shown online, but it suffers from a critical bug. If you change the screen orientation while the thread is active, the UI fails to receive notification when the thread completes. Here is what you see:



At this point the user has to exit and re-start the app, because the button will never again become enabled.

When the screen orientation changes, Android destroys the existing Activity instance and creates a completely new Activity as a replacement. This is all part of the Activity lifecycle mentioned before. This diagram shows what happens:



Although Android creates a new instance of `HomeActivity`, our background thread does not stop automatically. Even worse, our thread is an inner class with an implicit reference back to the `HomeActivity` instance. If the thread never stops, this causes a memory leak. But in our case, it means when the thread completes, it notifies the old `HomeActivity` instance instead of the new one. If you rotate the screen while our thread is running, **the GUI shows incorrect**

values.

These kinds of state management problems are not limited to screen rotation. When an incoming phone call arrives, Android pauses the `HomeActivity` to display the phone call. The user might also bring up the dialer or navigate to some other `Activity`. You need to test all of these scenarios to ensure the background thread completes successfully.

Fixing the Bug

There are a variety of things we can do to fix this bug. Some options include:

- Override `Activity.onPause()` and stop the thread. When the activity resumes, simply start another thread. While this would work, it is inefficient. So long as we started a thread, we may as well try to utilize the data it just fetched.
- Override `Activity.onRetainNonConfigurationInstance()` and return a reference to the running `Thread`. When Android creates the new `HomeActivity` instance, you can obtain the thread via `getLastNonConfigurationInstance()`. This is only a partial solution because our inner class thread still has a reference to the old `HomeActivity`. We'd need to make a static or top-level Thread class, and explicitly set a reference to the new `HomeActivity` instance after the rotation. This is possible, but tricky to get right.
- Use an Android `Service` along with `BroadcastReceiver`, but that's a pretty expensive solution. `BroadcastReceiver` is more appropriate when sending messages to other applications, and is overkill when sending messages within a single `Activity`.

Our Solution

Let's start by creating a data model that keeps track of the `HomeActivity` state.

HomeModel.java

```
/**
 * Data model for the HomeActivity. Fires events when the data
 * fetch thread begins and ends.
 */
public class HomeModel implements Serializable {
  private static final long serialVersionUID = 1L;

  public enum State { NOT_STARTED, RUNNING, FINISHED }

  private State state = State.NOT_STARTED;
  private HomeModelListener homeModelListener;

  public synchronized void setHomeModelListener(HomeModelListener l) {
    homeModelListener = l;
  }

  public void setState(State state) {
    // copy to a local variable inside the synchronized block
    // to avoid synchronization while calling homeModelChanged()
    HomeModelListener hml = null;
    synchronized (this) {
      if (this.state == state) {
        return; // no change
      }
      this.state = state;
      hml = this.homeModelListener;
    }
    // notify the listener here, not synchronized
    if (hml != null) {
      hml.homeModelChanged(this);
    }
  }

  public synchronized State getState() {
    return state;
  }
}
```

HomeModelListener.java

The `HomeModelListener` interface is trivial.

```
public interface HomeModelListener {
  void homeModelChanged(HomeModel hm);
}
```

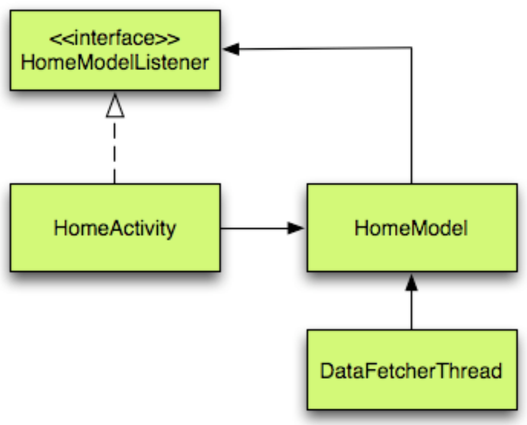
DataFetcherThread.java

Next, we'll break out the thread into a standalone class. This eliminates the problem with our inner class holding an implicit reference to the enclosing `HomeActivity` instance.

```
public class DataFetcherThread extends Thread {
  private final HomeModel homeModel;

  public DataFetcherThread(HomeModel homeModel) {
    this.homeModel = homeModel;
  }

  public void start() {
    homeModel.setState(HomeModel.State.RUNNING);
    super.start();
  }

  public void run() {
    try {
      TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
    } finally {
      homeModel.setState(HomeModel.State.FINISHED);
    }
  }
}
```

Class Diagram

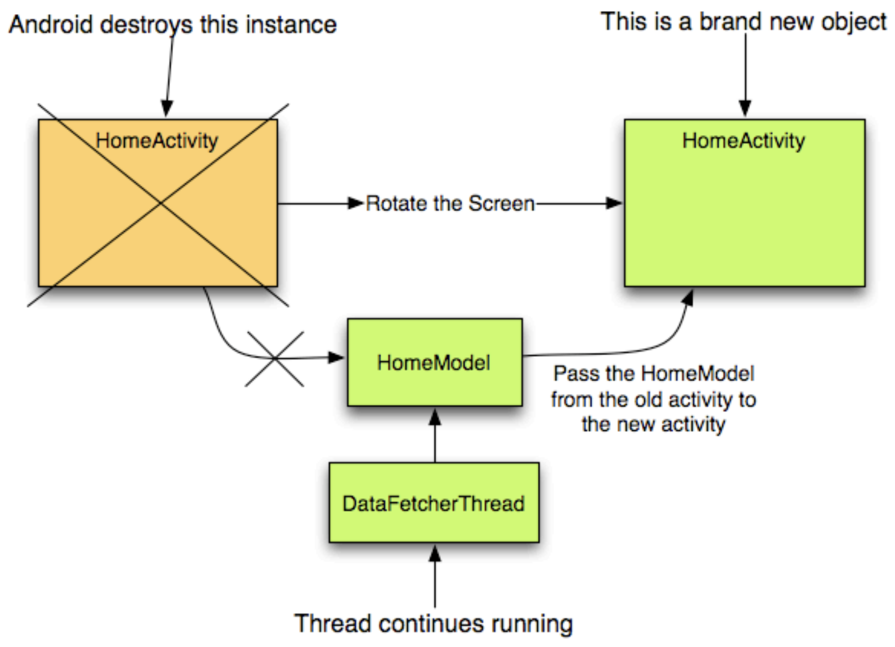
Here is a diagram that shows how these pieces fit together.



- `HomeActivity` creates `DataFetcherThread`, but does not retain a reference to it.
- `HomeActivity` implements `HomeModelListener`, updating the button and label display whenever the data model changes.
- The thread is no longer an inner class, so it retains no reference back to `HomeActivity`.

Improved Rotation

What happens when the user rotates the screen? When Android destroys the first `HomeActivity` instance, the activity saves the `HomeModel` reference in its saved instance state. It also detaches itself as a listener. When Android creates the second `HomeActivity`, it obtains the saved `HomeModel` instance from the saved instance state. Hopefully this diagram clarifies a bit:



Android destroys this instance
This is a brand new object
HomeActivity
Rotate the Screen
HomeActivity
HomeModel
Pass the HomeModel from the old activity to the new activity
DataFetcherThread
Thread continues running

HomeActivity, Final Version

Here is *HomeActivity.java* in its entirety, using all of the changes mentioned above.

```
public class HomeActivity extends Activity implements OnClickListener, HomeModelListener {

  private Button mStartButton;
  private TextView mStatusLabel;

  private static final String ENABLED_KEY = "com.ociweb.buttonEnabled";
  private static final String HOME_MODEL_KEY = "com.ociweb.homeModel";

  // background threads use this Handler to post messages to
  // the main application thread
  private final Handler mHandler = new Handler();

  // this data model knows when a thread is fetching data
  private HomeModel mHomeModel;

  // post this to the Handler when the background thread completes
  private final Runnable mUpdateDisplayRunnable = new Runnable() {
    public void run() {
      updateDisplay();
    }
  };

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.home);

    mStartButton = (Button) findViewById(R.id.start_background_thread_btn);
    mStatusLabel = (TextView) findViewById(R.id.thread_status_label);

    mStartButton.setOnClickListener(this);

    if (savedInstanceState != null) {
      if (savedInstanceState.containsKey(ENABLED_KEY)) {
        mStartButton.setEnabled(savedInstanceState.getBoolean(ENABLED_KEY));
      }
      if (savedInstanceState.containsKey(HOME_MODEL_KEY)) {
        mHomeModel = (HomeModel) savedInstanceState.getSerializable(HOME_MODEL_KEY);
      }
    }
    if (mHomeModel == null) {
      // the first time in, create a new model
      mHomeModel = new HomeModel();
    }
  }

  @Override
  protected void onPause() {
    super.onPause();
    // detach from the model
    mHomeModel.setHomeModelListener(null);
  }

  @Override
  protected void onResume() {
    super.onResume();
    // attach to the model
    mHomeModel.setHomeModelListener(this);

    // synchronize the display, in case the thread completed
    // while this activity was not visible. For example, if
    // a phone call occurred while the thread was running.
    updateDisplay();
  }

  public void homeModelChanged(HomeModel hm) {
    // this may be called from a background thread, so post
    // to the handler
    mHandler.post(mUpdateDisplayRunnable);
  }

  @Override
  protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putBoolean(ENABLED_KEY, mStartButton.isEnabled());
    outState.putSerializable(HOME_MODEL_KEY, mHomeModel);
  }

  public void onClick(View v) {
    if (v == mStartButton) {
      new DataFetcherThread(mHomeModel).start();
    }
  }

  private void updateDisplay() {
    mStartButton.setEnabled(mHomeModel.getState() != HomeModel.State.RUNNING);

    switch (mHomeModel.getState()) {
    case RUNNING:
      mStatusLabel.setText(R.string.thread_running);
      Toast.makeText(this, R.string.thread_running, Toast.LENGTH_SHORT);
      break;
    case NOT_STARTED:
      mStatusLabel.setText(R.string.thread_not_started);
      break;
    case FINISHED:
      mStatusLabel.setText(R.string.thread_finished);
      Toast.makeText(this, R.string.thread_finished, Toast.LENGTH_SHORT);
      break;
    }
  }
}
```

Summary

Understanding the Activity Lifecycle is critical to success with Android. Since mobile devices have limited resources, activities are constantly paused, resumed, initialized, and destroyed. Your application code has to handle these transitions with grace, which is probably the most challenging aspect of Android development.

Recommended Reading

- *Hello, Android* by Ed Burnette is probably the best book for programmers new to Android.
- Android Official Site - http://www.android.com
- Android Developer site on Google Code (SDK, documentation, and tutorials) - http://code.google.com/android/
- Android Open Source Project (source code for the OS, SDK, and Dalvik VM) - http://source.android.com/