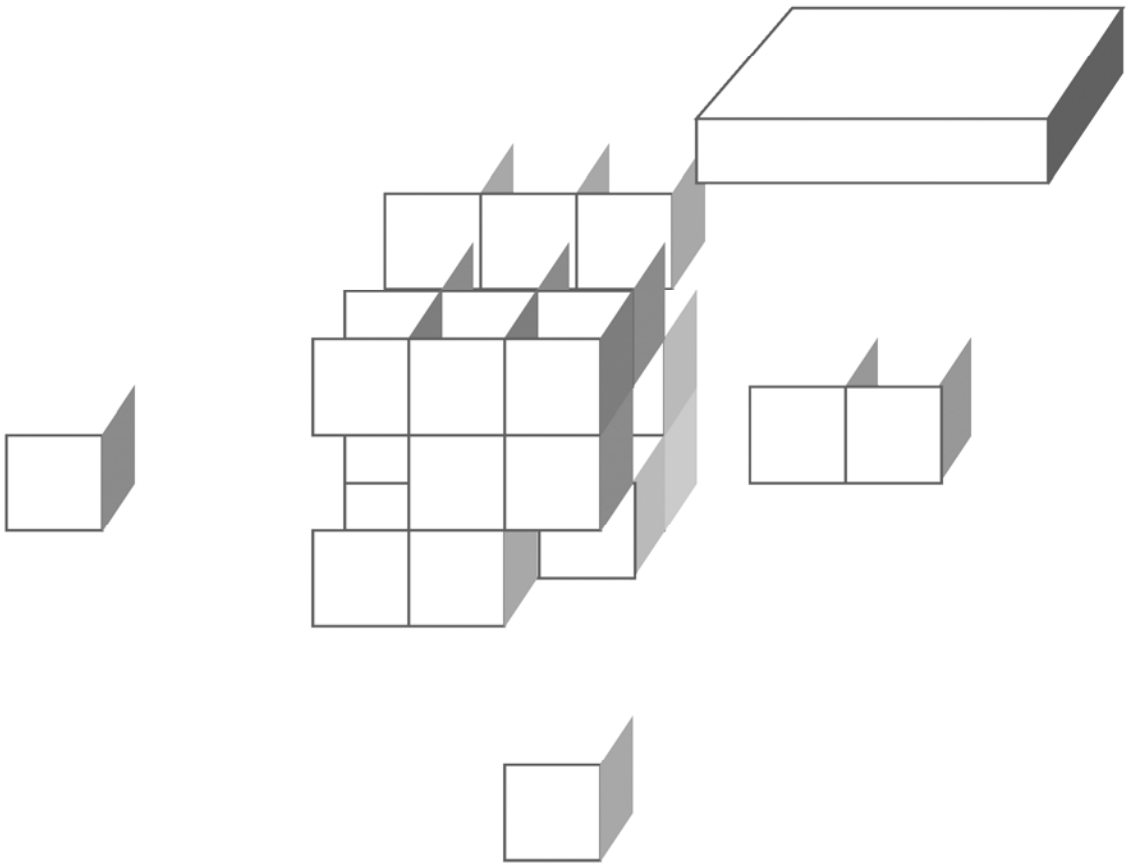
Chapter 15

Asynchronous Tasks



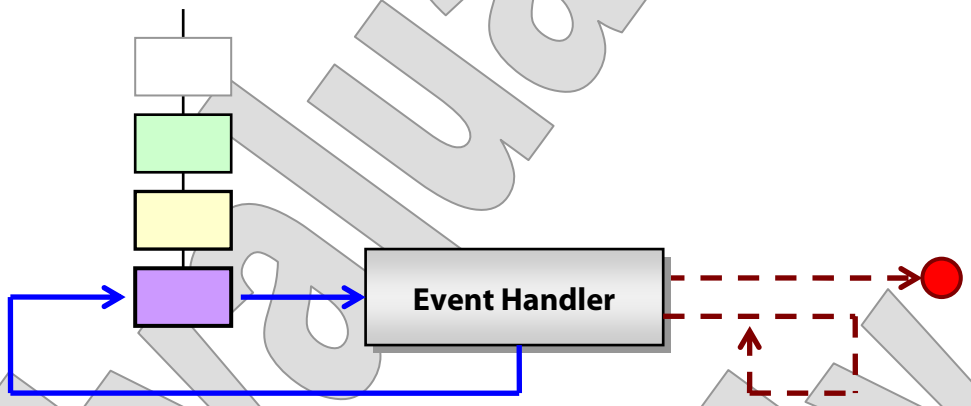
OBJECTIVES

After completing "Asynchronous Tasks," you will be able to:

- Describe the process and threading model of Android applications.
- Use **Looper** and **Handler** classes to post messages to threads, including the main application thread for updates to the UI.
- Use the **AsyncTask** class to perform longer-running tasks on separate threads while still interacting with the UI at certain points in the task lifecycle.
- Manage progress dialogs during task execution, and allow the user to cancel the task by canceling the dialog.
- Implement error handling over asynchronous tasks.

Multi-Threading in Android

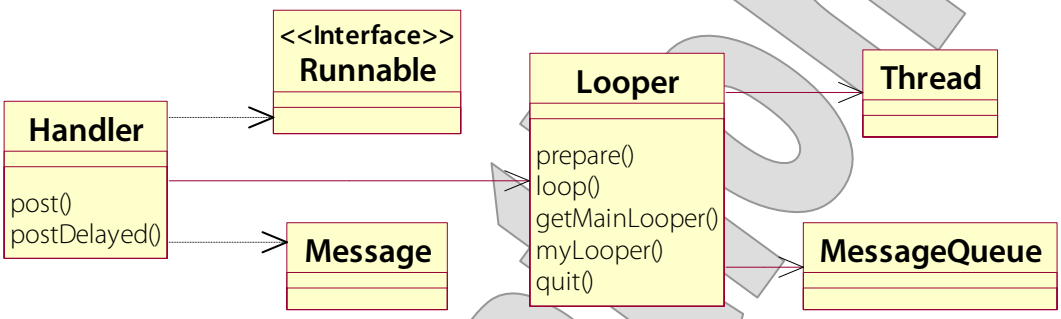
- **Android implements the familiar Java SE model of threads and thread groups, but the structure of groups and threads it sets up is unique to the Android OS, as are some rules it puts in place:**
  - An application occupies a **process** on the OS.
  - The process starts a **main thread** for the application.
  - This thread is responsible for all **user-interface** actions – including responding to user gestures.



  - There is a **message loop** processed by this thread, on which all user actions are encountered, and other events such as invalidating a view are placed here as well.
  - Applications may create their own **worker threads** – but there are important restrictions on what they can do.
  - Specifically, only the thread that creates a **View** can operate on that **View**. That means that spawned threads cannot **update the UI**!

Looper and Handler

- The key types to understanding Android's processing model are **Looper** and **Handler**, found in the **android.os** package.



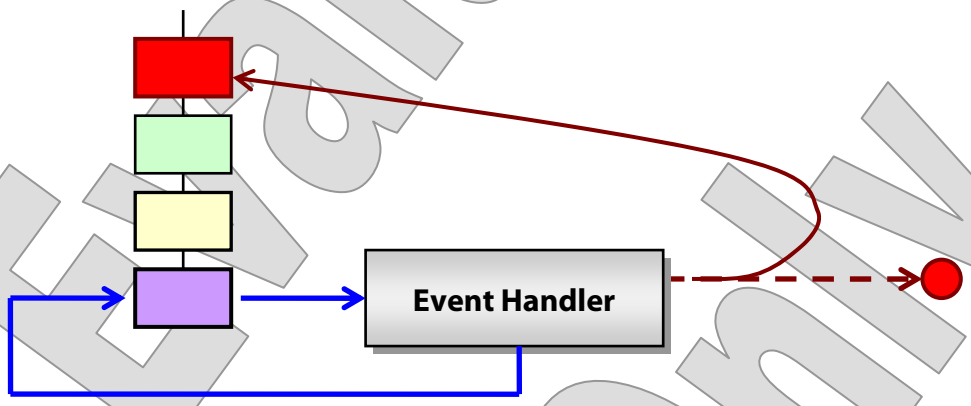
- A **Looper** implements a message-handling loop for a thread.
  - Any thread can have a looper – call **prepare** to create a looper for the current thread of execution, and **loop** to start processing.
  - The main thread has one, generally known as the **main looper**, and you can get this by calling the static method **getMainLooper**.
- A **Handler** attaches to a **Looper** and provides a simple means of putting messages on the looper's message queue.
  - Use **post**, **postDelayed**, or one of a few other variants that give various options for the timing of message enqueueing.
- This makes **Handler** the easiest way for one thread to assign a tasks to another – the recipe is basically this:

```
Handler handler =
  new Handler (Looper.getMainLooper ());
handler.post (myRunnable);
```

  - The **Runnable** will be placed on the queue of the main thread, and executed at some future time.

Background Tasks

- Let's say you want to carry out a longer-running task in response to some user gesture, such as a button click.
- You could implement a **Runnable** and start a **Thread** to call your **run** method – and as long as the activity in question never needs to touch the user interface, that will work fine.
- If you need to report your results or progress to the user, though, you will find that any attempt to operate on the user interface will result in an exception.
- So you will instead need to signal the main thread that it should carry out some UI updates – but how to do this?
- This is where a **Handler** is quite useful.



  - When your worker thread is done processing, it can post a **Runnable** to the main looper.
  - The code is executed on the main thread and so is free to update the UI.

The AsyncTask Class

- **This is a reasonably simple structure, but it makes for a lot of repetitive and error-prone coding.**
- **Android also provides a very nice encapsulation of this common usage, in the AsyncTask class.**
  - Public methods let a caller manage task execution:

```
public void execute (Params...);
public void cancel (boolean);
```

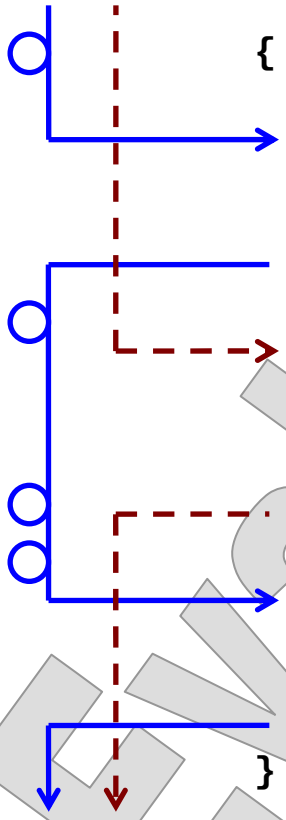
  - At first glance the protected methods of this class just seem to break down some complex task into phases:

```
* protected void onPreExecute ();
  protected Result doInBackground (Params...);
* protected void onPostExecute (Result);
* protected void onProgressUpdate (Progress...);
  protected void publishProgress (Progress...);
* protected onCancelled ();
* protected onCancelled (Result);
```

  - The genius of this class is that it assures that certain methods will be called on the **UI thread**, while some will be called on a separate worker thread.
  - Those marked * above are called on the UI thread.
- **The class code does all the heavy lifting in the background, such as scheduling execution and posting messages to the UI thread to assure that the right code is called on the right thread at the right time.**

The AsyncTask Class

- So if you were to implement a longer-running task in this way, and launch it from the UI thread, your methods would be called in sequence, but on different threads as appropriate:



```
public class MyTask
  extends AsyncTask<Void,Void,Void>
{
  protected void onPreExecute ()
  {
    ...
  }

  protected Void doInBackground
    (Void... params)
  {
    ...
  }

  protected void onPostExecute
    (Void result)
  {
    ...
  }
}
```

- Similar coordination occurs over other protected methods:
  - You can **cancel** from any thread, but **onCancelled** will be called on the UI thread.
  - You can **publishProgress** from **doInBackground** – so, on the worker thread – but **onProgressUpdate** will be called on the UI thread, for example to refresh progress bars or other indicators.

The AsyncTask Class

- **One of the trickier aspects to using AsyncTask is that it is parameterized – three times!**

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

- **Params defines the type of parameters that can be passed into the task.**
  - Pass any number of arguments of this type to **execute**.
  - Your arguments will be passed along to **doInBackground**.
- **Progress defines the type of information that can be posted from the background task as an indication of progress.**
  - Code in **doInBackground** will pass arguments of this type in any calls it makes to **publishProgress**.
  - These will be handed over to the **onProgressUpdate** method.
- **Result defines a result type for the task.**
  - This will be returned by **doInBackground**.
  - It will be passed to **onPostExecute**.
- **AsyncTasks can not be reused – execute and throw it away.**
- **A final note about this class – it is intended for tasks that are not very long in duration.**
  - Long enough, perhaps, for it to be worth showing the user some progress indication.
  - But for tasks taking "more than a few seconds," to quote the API documentation, should be handled by Java SE concurrent types such as **ExecutorService** – or implemented in Android services.

A Running Clock

DEMO

- **We'll look at a simple example of multi-threading in the Quiz application: we will put a running clock on the screen so the user can see elapsed time.**
  - The starter code has traditional Java **Timer** and **TimerTask** code to update the clock every quarter-second.
  - We will see that this doesn't mesh well with Android UIs, and refactor it to use **AsyncTask**.
- **Do your work in Demos/Async.**
  - The completed demo is found in **Examples/Quiz/Step5**.
- See **src/cc/quiz/QuestionActivity.java.**
  - There is a field to keep track of the running timer:

```
private Timer timer;
```

A Running Clock

DEMO

– We hook resume/pause lifecycle events to start/stop the timer. We create the timer task with reference to a text view, which it should update with the formatted elapsed time, and a starting time from which to calculate:

```
@Override
public void onResume ()
{
  super.onResume ();

  timer = new Timer ();
  timer.schedule (new Stopwatch
    ((TextView) findViewById (R.id.clock),
    ((Quiz) getApplication ()).getStartTime ()),
      250, 250);
}

@Override
public void onPause ()
{
  super.onPause ();
  timer.cancel ();
}
```

A Running Clock

DEMO

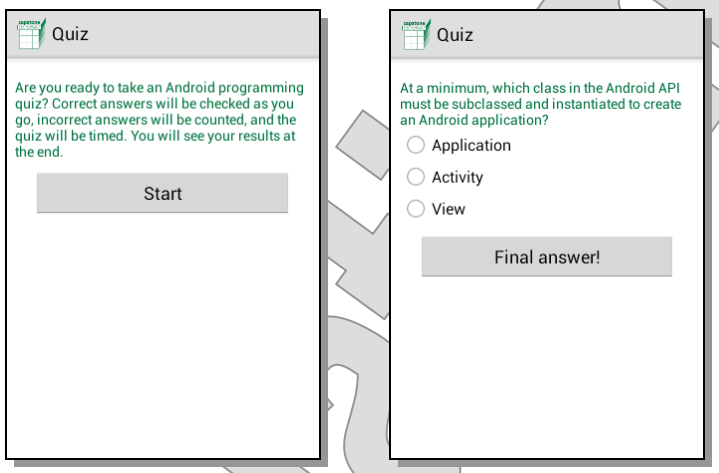
– Our timer task does the calculation, and for the moment just logs that it's being called:

```
private class Stopwatch
  extends TimerTask
{
  private TextView target;
  private long startTime;

  public Stopwatch
    (TextView target, long startTime)
  {
    this.target = target;
    this.startTime = startTime;
  }

  public void run ()
  {
    long testTime =
      SystemClock.elapsedRealtime () - startTime;
    int minutes = (int) (testTime / 60000);
    int seconds = (int)
      (testTime % 60000) / 1000;

    String formatted = String.format
      ("%d:%02d", minutes, seconds);
    Log.i ("Quiz", formatted);
    //TODO target.setText (formatted);
  }
}
```

A Running Clock

DEMO

1. Test the starter application. When you click **Start** and get to the first question, you will start to see those log messages in the LogCat view.




```
Quiz          0:01
Quiz          0:01
Quiz          0:01
Quiz          0:01
Quiz          0:02
Quiz          0:02
Quiz          0:02
Quiz          0:02
Quiz          0:03
...
```

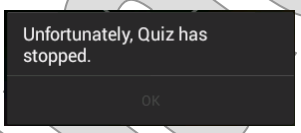
A Running Clock

DEMO

2. Try un-commenting the final line of the **run** method, so that it will show the elapsed time:

```
        String formatted = String.format
          ("%d:%02d", minutes, seconds);
        Log.i ("Quiz", formatted);
        target.setText (formatted);
      }
    }
```

3. Test again ... whoops!




```
E/AndroidRuntime(5702): android.view.ViewRootImpl
  $CalledFromWrongThreadException: Only the
  original thread that created a view hierarchy can
  touch its views.
```

A Running Clock

DEMO

4. Change the base class of **Stopwatch** from **TimerTask** to **AsyncTask**, with **Void** for all the type arguments:

```
private class Stopwatch
  extends AsyncTask<Void,Void,Void>
```

5. Now you will have to do your own timing. Implement the method **doInBackground** to sleep for a quarter-second at a time, as long as it has not been canceled:

```
@Override
protected Void doInBackground (Void... params)
{
  while (!isCancelled ())
    try
    {
      Thread.sleep (250);
    }
    catch (InterruptedException ex) {}

  return null;
}
```

6. Now, you could call your **run** method from inside this loop – but you would not have gained anything, because **doInBackground** runs on a non-UI thread and you would still be prohibited from adjusting the UI state from this thread. Instead, change the signature of the method to be an override of **onProgressUpdate**:

```
~~public void run ()~~
@Override
protected void onProgressUpdate (Void... values)
```

A Running Clock

DEMO

7. Now, call **publishProgress** from the **doInBackground** method:

```
while (!isCancelled ())
  try
  {
    publishProgress (params);
    Thread.sleep (250);
  }
```

8. Now, instead of a **Timer**, keep a field of type **Stopwatch** itself:

```
private Stopwatch stopwatch;
```

9. In **onResume**, the creation of the **Stopwatch** instance is unchanged. But now, you capture a reference to it, directly, and execute it:

```
@Override
public void onResume ()
{
  super.onResume ();

  stopwatch = new Stopwatch
    ((TextView) findViewById (R.id.clock),
    ((Quiz) getApplication ()).getStartTime ());
  stopwatch.execute ();
}
```

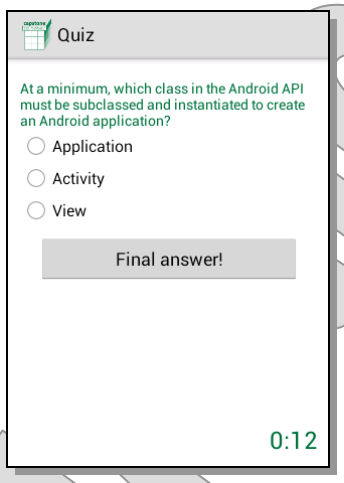
10. Cancel the task, not the timer, in **onPause**:

```
@Override
public void onPause ()
{
  super.onPause ();
  stopwatch.cancel (true);
}
```

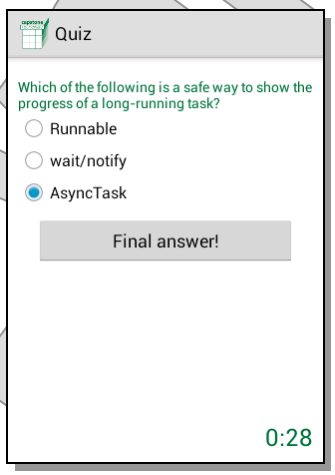
A Running Clock

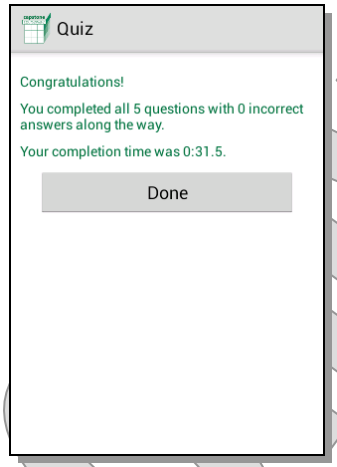
DEMO

11. Run the application and see that the call to **target.setText** is now safe, because the necessary thread synchronization is handled for you by **AsyncTask** and this call is being made on the UI thread.



– And now, you should have no trouble answering the new, final question of the quiz ...





Using the ProgressDialog

- **That's great, but what really have we gained?**
  - Possibly some better performance from multi-threading.
  - Possibly some better responsiveness as the user thread is able to go back to processing user gestures a bit sooner.
- **But the big point of asynchronous tasks is combining those advantages with the ability to do UI updates ...**
  - **Before** the task starts – that's **onPreExecute**
  - **During** the task – that's **onProgressUpdate**
  - **After** the task completes – that's **onPostExecute**
- **So a common strategy is to use all three of these methods to show the user the progress of the task in real time:**
  - Show a **ProgressDialog** at the start. This might just show a "working" animation, but it could also show a more meaningful **progress bar** to tell the user how far along we've gotten.
  - **Update** the progress bar as the task publishes progress updates.
  - **Close the dialog** when the task is complete.
  - Make the dialog **cancelable**, and if canceled take this as a signal to cancel the task itself.

Downloading Scribbles

EXAMPLE

- In **Examples/Scribble/Step9**, the tasks related to cloud storage have been refactored to use threading via **AsyncTask**.
- See **src/cc/draw/android/ScribbleActivity.java**.
  - In the **LoadCloud** menu handler, **onOptionsItemSelected** starts an activity to offer the user a directory of downloadable drawings from the Lockbox storage service.
  - Then, the **onActivityResult** method takes the selected name and kicks off a request to the web service and, when done, updates the window title and invalidates the content view.
  - It did this before, too – but now it happens by way of an asynchronous task, **LoadCloudTask**:

```
public void onActivityResult (int requestCode,
  int resultCode, Intent intent)
{
  ... // defensive code / error handling

  name = extras.getString
    (CloudBrowser.EXTRA_SELECTED_URL);
  new LoadCloudTask ().execute
    (view.getModel ());
  }
}
```

Downloading Scribbles

EXAMPLE

– The task class accepts a drawing as an input, and provides a boolean result:

```
private class LoadCloudTask
  extends AsyncTask<List<Element>,Void,Boolean>
  implements OnCancelListener
{
```

– It also implements **OnCancelListener**, so that it can respond to the user should the user choose to cancel the task in-progress.

– It shows a progress dialog before starting its work:

```
  ProgressDialog dialog;

  @Override
  protected void onPreExecute ()
  {
    dialog = new ProgressDialog
      (ScribbleActivity.this);
    dialog.setMessage (getResources ()
      .getText (R.string.cloud_load_description));
    dialog.setCancelable (true);
    dialog.setOnCancelListener (this);
    dialog.show ();
  }
```

Downloading Scribbles

EXAMPLE

– **doInBackground** does little more than delegate to **loadURL** as we've seen it in earlier versions of the application:

```
@Override
protected Boolean doInBackground
  (List<Element>... params)
{
  List<Element> model = params[0];

  String requestURL =
    getServiceURL () + "/" + name;
  Log.i ("Storage", requestURL);

  return loadURL (requestURL, model);
}
```

– **onPostExecute** cleans up the dialog, and does other post-processing that was done in **onActivityResult** before:

```
@Override
protected void onPostExecute (Boolean success)
{
  dialog.dismiss ();
  if (success)
  {
    setTitle (getResources ().getText
      (R.string.app_name) + ": " + name);
    view.invalidate ();
  }
  else ...
}
```

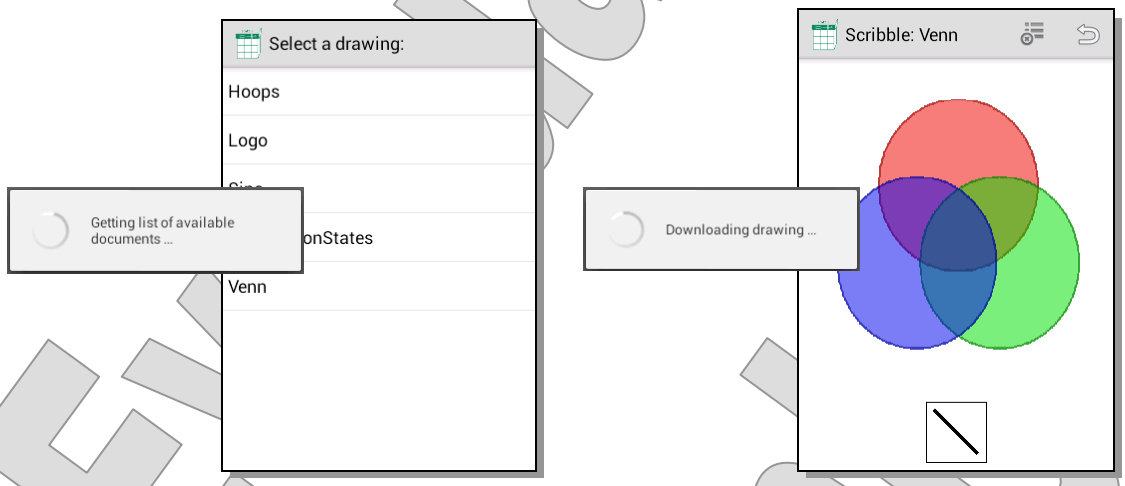
Downloading Scribbles

EXAMPLE

  - Finally, we catch cancellation events from the progress dialog, so that we can cancel the running task if the user cancels the dialog:

```
  public void onCancel (DialogInterface dialog)
  {
    cancel (true);
  }
}
```

- **Test, and see the progress dialog for a few seconds before the download of available drawings, and then again during the drawing download itself:**



- **The application is multi-threaded in two more places:**
  - For the **SaveCloud** handler, which now uses a **SaveCloudTask**
  - In the **CloudBrowser**, so that the act of getting the list of available drawings from the service is also threaded and cancelable

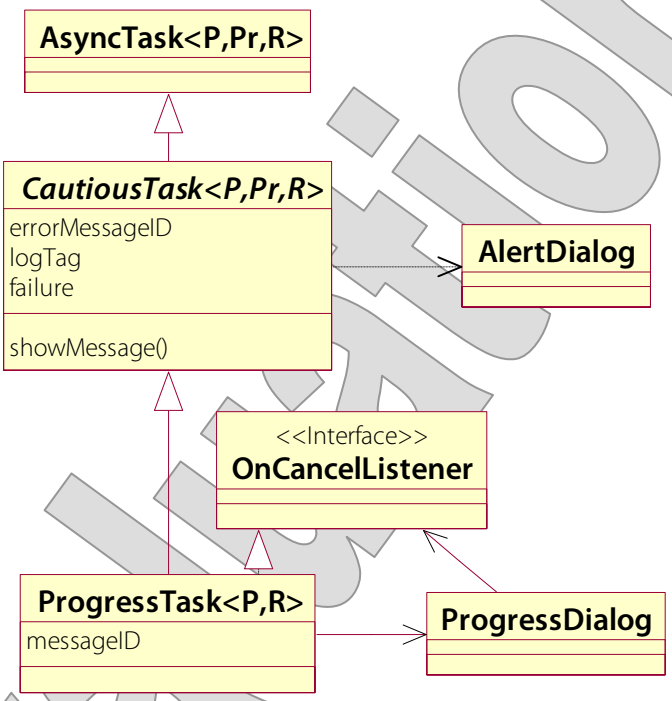
Error Handling

- **Special care must be taken to handle errors thrown during execution of an asynchronous task.**
- **Exceptions will naturally fall through different method call stacks on different threads; and some of those involve your calling code, but some do not.**
- **By default, exceptions encountered in various methods will behave as follows:**
  - An exception in **onPreExecute** occurs on the **UI thread** and will fall through to the caller of **execute** on the task.
  - An exception in **doInBackground** occurs on the **background thread** and will fall through that thread's call stack – which does not include any application methods outside of the task class.
  - An exception in **onPostExecute** occurs on the **UI thread** but by way of a queued message, and so will not fall through to your calling code.
- **So it is important generally to assure that an asynchronous task does its own error handling.**
- **And of course any handling that involves the UI must occur on the UI thread! so some signaling between doInBackground and onPostExecute is often indicated.**

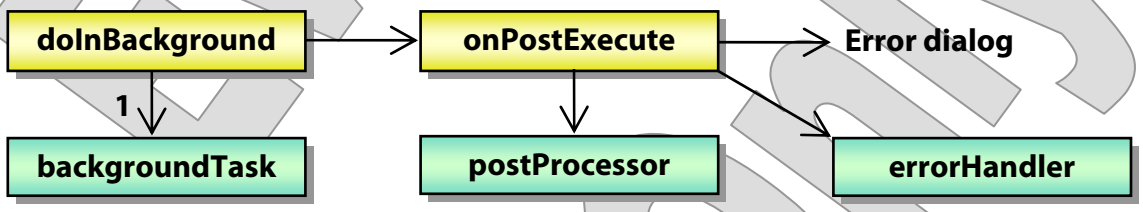
ProgressTask and CautiousTask

EXAMPLE

- In **Examples/Insurance/Step16**, there are two encapsulations of asynchronous tasks with specific feature sets:



- **CautiousTask** overrides **doInBackground** to call its own helper method, and checks for failures:



  - Then it overrides **onPostExecute** either to delegate to another helper method for normal post-processing, or to show an error dialog and delegate to a different helper for error handling.

- **ProgressTask** refines this by showing a progress dialog from **onPreExecute** to **onPostExecute**.

ProgressTask and CautiousTask

EXAMPLE

- See **src/cc/android/CautiousTask.java.**

```
public abstract class
    CautiousTask<Params,Progress,Result>
  extends AsyncTask<Params,Progress,Result>
  implements OnDismissListener
```

  - It defines three helper methods: one to be called by **doInBackground**, and two to be called by **onPostExecute**:

```
protected abstract Result
  backgroundTask (Params... params);
protected void postProcessor (Result result) {}
protected void errorHandler (Throwable failure) {}
```

  - It requires error-message and logging information through its constructor signature:

```
public CautiousTask (Activity parent,
  int errorMessageID, String logTag)
{
  this.parent = parent;
  this.errorMessageID = errorMessageID;
  this.logTag = logTag;
}
```

ProgressTask and CautiousTask

EXAMPLE

– It overrides **doInBackground** in order to catch any exceptions. It logs the caught exception, and stores it as the **failure** field:

```
@Override
protected Result doInBackground
  (Params... params)
{
  try
  {
    return backgroundTask (params);
  }
  catch (Throwable e)
  {
    Log.e (logTag, "An error occurred ...", e);
    failure = e;
  }

  return null;
}
```

– It then overrides **onPostExecute** to invoke **postProcessor** or to show an error message to the user, depending on the success of the operation:

```
@Override
public void onPostExecute (Result result)
{
  if (wasSuccessful ())
    postProcessor (result);
  else
    showMessage ();
}
```

ProgressTask and CautiousTask

EXAMPLE

- **showMessage** just shows the appropriate **AlertDialog**.

```
protected void showMessage
  (int messageID, boolean thenCallErrorHandler)
{
  AlertDialog dialog =
    new AlertDialog (parent, messageID);

  if (thenCallErrorHandler)
    dialog.setOnDismissListener (this);

  dialog.show ();
}
```

- But we register a dialog-dismiss listener, so that after the user has seen the error message, the **errorHandler** helper method will be invoked to allow additional handling.

```
public void onDismiss (DialogInterface dialog)
{
  errorHandler (failure);
}
```

- **So a subclass can work from here:**
  - It must implement **backgroundTask**, as it would normally have implemented **doInBackground**.
  - It can then implement normal post-processing in **postProcessor**, and additional error handling in **errorHandler**.

ProgressTask and CautiousTask

EXAMPLE

- See **src/cc/android/ProgressTask.java**; the contributions of this class are simpler and mostly include techniques we've already seen.

```
public abstract class ProgressTask<Params,Result>
  extends CautiousTask<Params,Void,Result>
  implements OnCancelListener
```

  - For starters, notice that we squeeze out the **Progress** type parameter. Since this class only shows an activity animation in its progress dialog, there is nothing useful we could say with a call to **publishProgress**. So we simplify here.
  - It captures an additional **messageID** in its constructor, for use in the progress dialog.
  - It overrides **onPreExecute** to show the progress dialog:

```
@Override
protected void onPreExecute ()
{
  dialog = new ProgressDialog
    (parent, messageID);
  dialog.setCancelable (true);
  dialog.setOnCancelListener (this);
  dialog.show ();
}
```

ProgressTask and CautiousTask

EXAMPLE

- It overrides **onPostExecute** to dismiss the progress dialog, and then calls the base class implementation – which is the code in **CautiousTask** that forks between normal processing and error handling.

```
@Override
public void onPostExecute (Result results)
{
  dialog.dismiss ();
  super.onPostExecute (results);
}
```

- Finally it implements the cancellation listener, to cancel the task if the user requests it – just as we did in the Scribble demonstration earlier:

```
public void onCancel (DialogInterface dialog)
{
  cancel (true);
}
```

- **You will use both of these classes in the upcoming lab as you address threading concerns for the Insurance application and its network interactions.**

Threading in the Insurance Application LAB 15

Suggested time: 30-45 minutes

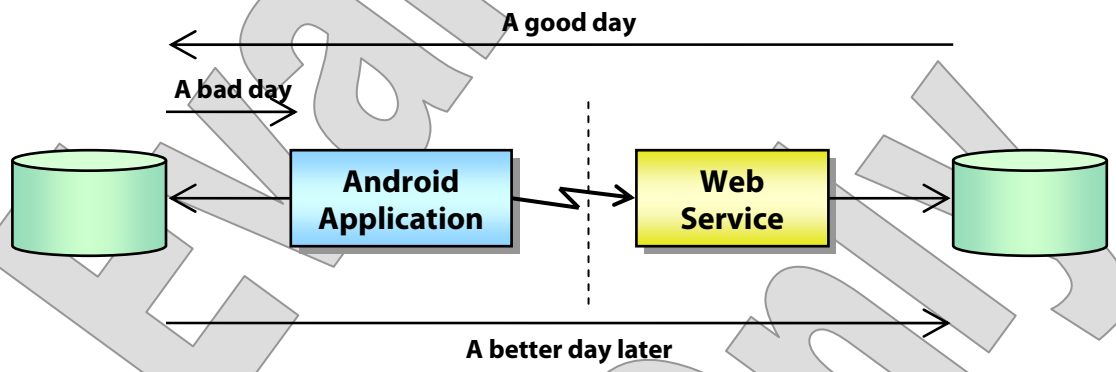
In this lab you will wrap existing upload and download operations in asynchronous tasks, taking advantage of the additional encapsulations already in the code.

Detailed instructions are found at the end of the chapter.

Local Backup and Synchronization

EXAMPLE

- In **Examples/Insurance/Step18**, we have followed the direction of this chapter's labs and taken similar steps to realize a robust, connected version of the application.
- This is the answer code to an additional lab from the full intermediate course, and it has these additional features:
  - It implements inner-class **AsyncTask**s for all of its download and upload operations: creating, modifying, and completing adjustment records; reporting missed appointments and requesting rescheduled appointments.
  - It saves a **local backup** of its most recent schedule download, in internal storage, and falls back to this local copy if started when the network is not available:



  - It **synchronizes** any changes made to the local copy if started when the network is available – and prior to doing a fresh download of the schedule.
- Full coverage is beyond our scope, but you may want to look at the code for this step to see the strategies; just about everything mentioned above is implemented in the **Application** class.

SUMMARY

- **Android threading is not fundamentally all that different from Java threading.**
- **The most obvious difference is the requirement that a single thread be responsible for all user interactions and all contact with the view hierarchies of activities.**
- **This in turn requires inter-thread communication.**
- **Then Android facilitates that communication with message queues for threads, and the Looper and Handler classes.**
- **It further simplifies the programming model for common cases with the AsyncTask class.**
  - **onPreExecute** and **onPostExecute** methods run on the thread that executes the task (usually the main thread).
  - **doInBackground** carries out the task on a separate thread.
  - Progress updates are facilitated so that they can be shown to the user with code that runs on the main thread as well.
- **Error handling in an asynchronous task is especially delicate and important, because it is hard or impossible to manage this from the code that executes the task.**

Threading in the Insurance Application LAB 15

In this lab you will wrap existing upload and download operations in asynchronous tasks, taking advantage of the additional encapsulations already in the code.

Lab workspace: **Labs/Lab15**

Backup of starter code: **Examples/Insurance/Step16**

Answer folder(s): **Examples/Insurance/Step17**

Files: **src/cc/insurance/android/Application.java**
src/cc/insurance/android/MainActivity.java
src/cc/insurance/android/fragment/Adjustment.java

Instructions:

1. Open **Application.java** and find the **updateAdjustment** method. You'll be refactoring this code into an asynchronous task that can be executed directly by the **AdjustmentActivity**.
2. Notice also the inner interface **OnUpdateComplete**. The method **onUpdateComplete** will be called when your upload completes successfully.
3. Create a new inner class **AdjustmentUpdater** that extends **ProgressTask**. Your type arguments can both be **Void**. We will do this a lot because as we implement our tasks as inner classes they will have a lot of outer-class state at their disposal and so won't need to pass information between threads using parameter and result types.
4. Give the class a field **listener**, of type **OnUpdateComplete**.
5. Give the class a constructor that takes an **Activity** and a reference to **OnUpdateComplete** as parameters. Call the superclass constructor, passing the activity, the string ID **progress_update_adjustment**, the constant **ERROR_MESSAGE_ID** (which we'll use as a general-purpose error message throughout the application code pretty soon), and the constant **LOG_TAG** (same story here). Now the base class is ready to show both progress and alert dialogs at appropriate times, with appropriate messages.
6. Store the **OnUpdateComplete** parameter in your **listener** field.

Threading in the Insurance Application LAB 15

7. The existing method has basically this structure: the first three lines set up and send an HTTP request; if it succeeds, there is a block of code that adjusts the data model and shows a notification; and if it fails we show an alert dialog. The first three lines of code, then, are appropriate as our background task; but the last will need to be invoked on the UI thread. That "success block" in-between will actually lose its toast notification, since we're showing a progress dialog already; but we will refactor it along with the error-handling code.

   So copy those first three lines into a new override of **backgroundTask** in your new inner class. But now, if the method fails, throw a new **RuntimeException** with a message that says so. In the case of success you don't need to do anything here.

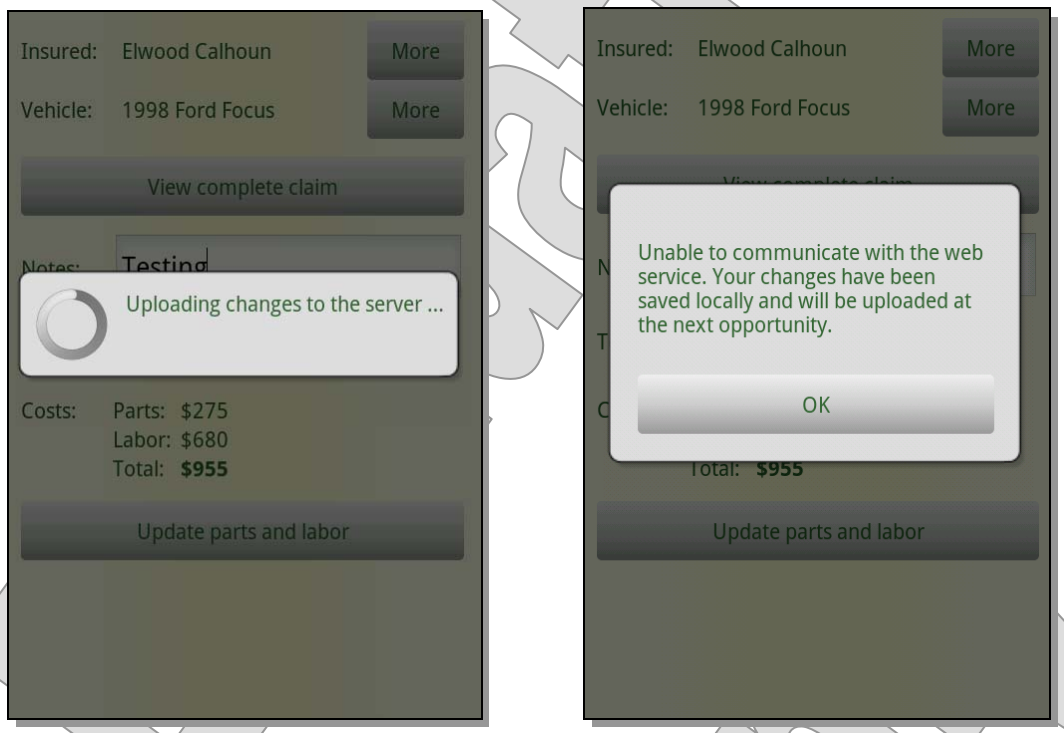
8. Return **null** to finish this method.
9. Override **postProcessor**. You'll need to start this method out with the same first two lines of code as **backgroundTask**, so that you're ready to work with the adjustment record and JSON representation.
10. Now you can copy all the "success" code from the existing **updateAdjustment** method – except you can leave out the call to **Toast.makeText**.
11. Instead, at the bottom of the method, check to see that **listener** is not **null**, and then call **listener.onUpdateComplete**.
12. Open **Adjustment.java** and find the call to **updateAdjustment** from the **save** method. Change this to instantiate an **AdjustmentUpdater**, passing the owning activity and the **listener** that's given to this method as a parameter. Then call **execute** on that object:

```
getMyApplication ().new AdjustmentUpdater
    (getActivity (), listener).execute ();
```

Threading in the Insurance Application LAB 15

13. Test this by running the application and selecting an appointment with an existing adjustment record. Open the adjustment record. Make any change – just a tweak to the notes field will be enough to see if the change is saved – and choose **Save** from the options menu.

    Try this in both good and bad conditions. You can force a failure by stopping the web server after starting the mobile application and downloading the schedule, but before saving your change to the adjustment. You should see sensible results in both cases: the progress dialog quietly goes away when an upload succeeds, but an alert is shown when it fails:



    Notice that bit about saving the schedule locally and synchronizing it isn't really true, or at least not yet. At the end of this chapter we'll see a more complete implementation of this and the other upload tasks that will manage an offline mode and also attempt synchronization when starting the application.

Threading in the Insurance Application LAB 15

14. For now let's turn our attention to the main schedule download. First, clean out the old **updateAdjustment** method completely; it's dead code now.
15. Back in the **Application** class, create a new inner class **ScheduleLoader**. This too will be a **ProgressTask<Void,Void>**.
16. Give the class a boolean field called **initialDownload**. Ultimately this flag will affect the choice of message shown to the user on download failure, as we're going to support both a download on startup and a menu-driven download on demand.
17. Give your class a constructor that takes two parameters: a parent **Activity**, and a boolean parameter whose value you will store in the **initialDownload** field. Call the superclass constructor, more or less as you did in your **AdjustmentUpdater** class, but the two message IDs this time or **progress_load_schedule** and **error_no_data**. Then store off the **initialDownload** value.
18. Implement **backgroundTask**: start by calling **schedule.clear**, and then call **loadScheduleRemote** and return **null**.
19. Override **postProcessor** to call **fireScheduleChanged** when the download succeeds.
20. In the **onCreate** method of **MainActivity.java**, change the call to **loadScheduleRemotely** to be an instantiation of a **ScheduleLoader** (passing **this** and then **true** for the initial-download flag) and a call to **execute** the new task instance.
21. Test now, and you should see the asynchronous task running and giving clear indications of progress and error conditions, as with adjustment updates: