Multithreaded Programming Part II: Android-Specific Techniques

Originals of Slides and Source Code for Examples:
http://www.coreservlets.com/android-tutorial/

Customized Java EE Training: http://courses.coreservlets.com/
Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at your location.

For live Android training, please see courses at http://courses.coreservlets.com/.

Taught by the author of Core Servlets and JSP, More Servlets and JSP, and this Android tutorial. Available at public venues, or customized versions can be held on-site at your organization.

Courses developed and taught by Marty Hall
- JSF 2, PrimeFaces, servlets/JSP, Ajax, jQuery, Android development, Java 6 or 7 programming, custom mix of topics
- Ajax courses can concentrate on 1 library (jQuery, Prototype/Scriptaculous, Ext-JS, Dojo, etc.) or survey several
Courses developed and taught by coreservlets.com experts (edited by Marty)
- Spring, Hibernate/JPA, EJB3, GWT, Hadoop, SOAP-based and RESTful Web Services
Contact hall@coreservlets.com for details
Topics in This Section

- GUI update guidelines
- Updating UI after threads complete
- Downloading images from Internet
- Updating UI with View.post
- Updating UI with AsyncTask
Issues

• Nonresponsive GUI controls
  – Long-running code in main thread will make GUI controls nonresponsive.
• Threads cannot update user interface
  – Background threads are prohibited from updating UI.

Nonresponsive GUI Controls

• Problem
  – Long-running code in main thread will make GUI controls nonresponsive.
    • For example, if a Button’s onClick handler takes 5 seconds, then none of the other controls will respond for 5 seconds.

• Solution
  – Move time-consuming operations to background threads
    • This was the entire point of the previous lecture on the basics of multithreaded programming
Threads Cannot Update UI

• **Problem**
  – Background threads are prohibited from updating UI.
    • E.g., background threads cannot add a new View to the display, nor can they call setText on a visible View.

• **Solutions (alternatives)**
  – Wait until all threads are done, then update UI
    • When multithreading improves performance, but total wait time is small
  – Have background threads use View.post to send UI-updating operations back to UI thread
    • Especially when transitioning standard threading code to Android
  – Use AsyncTask to divide tasks between background and UI threads
    • Especially when developing for Android from beginning and you have a lot of incremental GUI updates.
Updating UI All at Once

• Scenario
  – Using threads improves performance.
  – Total wait time is small, so no need to display intermediate results.

• Approach
  – Add threads to ExecutorService task list.
  – Use awaitTermination to wait until all tasks are finished (or a timeout is exceeded)
  – Update UI. You are in UI thread, so this is safe/legal.

Example: Multi-URL Validator

• Idea
  – Let user enter any number of URLs
  – Check if they are legal by making HTTP HEAD requests and checking the server status line

• Approach
  – Make a List of URL result holders
  – For each URL, pass result holder to background thread. Background thread stores result in holder. Use inner class for background thread implementation.
  – After all tasks are complete or 10 seconds have elapsed (whichever is sooner), show results in ScrollView
Main Activity: Setup Code

```java
public class UrlCheckerActivity extends Activity {
    private LinearLayout mResultsRegion;
    private EditText mUrlsToTest;
    private int mGoodUrlColor, mForwardedUrlColor,
            mBadUrlColor;
    private Drawable mGoodUrlIcon, mForwardedUrlIcon,
            mBadUrlIcon;
    List<UrlCheckerResult> mResultsHolder =
        new ArrayList<UrlCheckerResult>();
    private final static int HEAD_TIMEOUT = 10;
    private float mResultTextSize;
    private int mResultPaddingSize;
```

Main Activity: Setup Code (Continued)

```java
@override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.url_checker);
    mUrlsToTest = (EditText)findViewById(R.id.urls_to_test);
    mResultsRegion = (LinearLayout)findViewById(R.id.results_region);
    Resources resources = getResources();
    mGoodUrlColor = resources.getColor(R.color.url_good);
    mForwardedUrlColor = resources.getColor(R.color.url_forwarded);
    mBadUrlColor = resources.getColor(R.color.url_bad);
    mGoodUrlIcon = resources.getDrawable(R.drawable.emo_im_happy);
    mForwardedUrlIcon = resources.getDrawable(R.drawable.emo_im_wtf);
    mBadUrlIcon = resources.getDrawable(R.drawable.emo_im_sad);
    mResultTextSize =
        resources.getDimension(R.dimen.url_checker_result_size);
    mResultPaddingSize =
        (int)resources.getDimension(R.dimen.url_checker_padding_size);
```
public void checkUrls(View clickedButton) {
    mResultsRegion.removeAllViews();
    mResultsRegion.requestLayout();
    String[] urls =
        mUrlsToTest.getText().toString().split("\\s+");
    ExecutorService taskList =
        Executors.newFixedThreadPool(50);
    for (String url: urls) {
        UrlCheckerResult resultHolder =
            new UrlCheckerResult(url);
        mResultsHolder.add(resultHolder);
        taskList.execute(new UrlTester(resultHolder));
    }
    try {
        taskList.shutdown();
        taskList.awaitTermination(HEAD_TIMEOUT,
            TimeUnit.SECONDS);
    } catch (InterruptedException e) {}}

for(UrlCheckerResult resultHolder: mResultsHolder) {
    UrlChecker result = resultHolder.getUrlResult();
    if (result == null) {
        result =
            new UrlChecker(resultHolder.getUrlString(),
                HEAD_TIMEOUT);
    }
    TextView displayedResult = new TextView(this);
    displayedResult.setTextColor(chooseColor(result));
    displayedResult.setTextSize(mResultTextSize);
    displayedResult.setText(result.toString());
    displayedResult.setCompoundDrawablesWithIntrinsicBounds
        (chooseIcon(result), null, null, null);
    displayedResult.setCompoundDrawablePadding
        (mResultPaddingSize);
    mResultsRegion.addView(displayedResult);
}
Main Activity: Runnable Inner Class

private class UrlTester implements Runnable {
    private UrlCheckerResult mResultHolder;

    public UrlTester(UrlCheckerResult resultHolder) {
        mResultHolder = resultHolder;
    }

    public void run() {
        UrlChecker result =
            new UrlChecker(mResultHolder.getUrlString());
        mResultHolder.setUrlResult(result);
    }
}

The run method does not update the UI directly. Instead, it updates a
data structure that will later be read and then used to update the UI
(code that does this was shown on previous slide).

The UrlChecker class does the actual network calls. Since different servers can
respond at different speeds, calling UrlChecker in a background thread can
substantially improve performance.

Main Activity: Helper Methods

private int chooseColor(UrlChecker urlResult) {
    if (urlResult.isGood()) {
        return(mGoodUrlColor);
    } else if (urlResult.isForwarded()) {
        return(mForwardedUrlColor);
    } else {
        return(mBadUrlColor);
    }
}

private Drawable chooseIcon(UrlChecker urlResult) {
    if (urlResult.isGood()) {
        return(mGoodUrlIcon);
    } else if (urlResult.isForwarded()) {
        return(mForwardedUrlIcon);
    } else {
        return(mBadUrlIcon);
    }
}

I use different text colors and different icons depending
on whether the URL is good, forwarded, or bad.
Main Helper Classes

• **UrlChecker**
  - Makes a HEAD request and parses status line. If status code is 301 or 302, looks up Location header.
  - Most of the code shown in earlier lecture on Networking

• **UrlCheckerResult**
  - Stores a URL and the associated UrlChecker result.
  - The timeout could be exceeded before the results are found, in which case the UrlChecker is null. In that scenario, we display a message that shows the URL and says that no result was found within the allotted time.

Results

Multithreading Examples

**URLs:**
- http://www.coreservlets.com/android-tutorialQ
- http://www.coreservlets.com/android-tutorial/Q

Check URLs

  - is GOOD: 200 -- OK
- http://www.coreservlets.com/android-tutorialQ
  - is FORWARDED to
  - http://www.coreservlets.com/android-tutorial/Q
  - is BAD: 404 -- Not Found
  - cannot be tested: Illegal URL format
Downloading Web Images

Customized Java EE Training: [http://courses.coreservlets.com/](http://courses.coreservlets.com/)
Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at your location.

Displaying a Network Image

**Idea**
- The decodeStream method of BitmapFactory can take an InputStream and produce a Bitmap. You can put a Bitmap into an ImageView

**Approach**
- Get an HttpURLConnection (as explained in second networking lecture)
- Get the input stream with connection.getInputStream()
- Pass the stream to BitmapFactory.decodeStream and store result in Bitmap
- Get a reference to an ImageView and pass new Bitmap to setImageBitmap
public class BitmapUtils {
    public static Bitmap bitmapFromUrl(String urlString) throws MalformedURLException, IOException {
        URL url = new URL(urlString);
        HttpURLConnection urlConnection = (HttpURLConnection)url.openConnection();
        InputStream in = urlConnection.getInputStream();
        Bitmap bitmapImage = BitmapFactory.decodeStream(in);
        urlConnection.disconnect();
        return (bitmapImage);
    }
}

**Making a View for an Image**

- **Idea**
  - Given an address, try to load a Bitmap from it. If successful, make an ImageView showing the Bitmap. If unsuccessful, make a TextView showing information on the error.

- **Approach**
  - Call BitmapUtils.bitmapFromUrl.
    - If no Exception, instantiate ImageView and pass Bitmap to setImageBitmap.
    - If Exception, instantiate TextView and store an error message in it.
**BitmapUtils: Getting a View for an Image**

```java
public static View viewForImage(String urlString, Context context) {
    Bitmap bitmapImage = null;
    String errorMessage = "";
    try {
        bitmapImage = BitmapUtils.bitmapFromUrl(urlString);
    } catch (MalformedURLException mue) {
        errorMessage = String.format("Malformed URL %s", urlString);
    } catch (IOException ioe) {
        errorMessage = String.format("Error downloading image: %s", ioe.getMessage());
    }
    if (bitmapImage != null) {
        ImageView displayedImage = new ImageView(context);
        displayedImage.setImageBitmap(bitmapImage);
        displayedImage.setPadding(5, 5, 5, 5);
        return(displayedImage);
    } else {
        TextView displayedMessage = new TextView(context);
        displayedMessage.setText(errorMessage);
        return(displayedMessage);
    }
}
```

**Example: Unthreaded Image Viewer**

- **Idea**
  - Let user enter URL of an image to be displayed
  - Either show that image, or show error message
- **Approach**
  - Read the address from EditText, pass to BitmapUtils.viewForImage
  - Add resultant View to the layout
public class SimpleImageDownloaderActivity extends Activity {
    private LinearLayout mMainWindow;
    private EditText mImageToDownload;
    private View mViewForImage;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_image_downloader);
        mMainWindow = (LinearLayout)findViewById(R.id.main_window);
        mImageToDownload = (EditText)findViewById(R.id.image_to_download);
        mViewForImage = BitmapUtils.viewForImage(imageUrl, this);
        mMainWindow.addView(mViewForImage);
    }

    public void downloadImage(View clickedButton) {
        if (mViewForImage != null) {
            mMainWindow.removeView(mViewForImage);
        }
        String imageUrl = mImageToDownload.getText().toString();
        mViewForImage = BitmapUtils.viewForImage(imageUrl, this);
        mMainWindow.addView(mViewForImage);
    }
}
**View.post**

- **Scenario**
  - Total wait time might be large, so you want to show intermediate results
  - You have a lot of normal threading code but you occasionally need to update UI.

- **Approach**
  - When background thread has result that should affect UI, create a Runnable and pass to post

```java
public void run() {
    // Background thread
doStuffButDontUpdateUI();
someView.post(new Runnable() {
    public void run() { updateUI(); }});
}
```
Example: Multithreaded Image Viewer (Version 1)

- **Idea**
  - Let user enter URLs of images to be displayed
  - Download and display them inside a scrolling horizontal LinearLayout

- **Approach**
  - Download images and associate them with ImageView, as shown in the previous subsection. If there was an error, create a TextView saying so.
  - Create a Runnable that will add the resultant View to the LinearLayout
  - Pass that Runnable to the `post` method of the LinearLayout

Main Activity: Setup Code

```java
public class ImageDownloader1Activity extends Activity {
    protected LinearLayout mResultsRegion;
    protected EditText mImagesToDownload;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.image_downloader);
        mImagesToDownload = (EditText)findViewById(R.id.images_to_download);
        mResultsRegion = (LinearLayout)findViewById(R.id.results_region);
    }
```
Main Activity: Button Handler

```java
public void downloadImages(View clickedButton) {
    mResultsRegion.removeAllViews();
    mResultsRegion.requestLayout();
    String[] images =
        mImagesToDownload.getText().toString().split("\s");
    addImages(images);
}

protected void addImages(String[] images) {
    ExecutorService taskList =
        Executors.newFixedThreadPool(50);
    for (String image: images) {
        taskList.execute(new ImageDownloader(image));
    }
}
```

The event handler is split into two methods because the next example will extend this class and override `addImages`, but will keep the inherited version of `downloadImages` unchanged.

Main Activity: Runnable Inner Class (For Background)

```java
private class ImageDownloader implements Runnable {
    private String mImageUrl;

    public ImageDownloader(String imageUrl) {
        mImageUrl = imageUrl;
    }

    public void run() {
        View viewToAdd =
            BitmapUtils.viewForImage(mImageUrl, ImageDownloader1Activity.this);
        mResultsRegion.post(new ViewAdder(viewToAdd));
    }
}
```

This is the task that the ExecutorService executes. It calls `viewForImage`, a time-consuming task that loads an image from the network and associates it with an `ImageView` (or a `TextView` if there was an error). The `View` is then sent to the main UI thread by putting it in a `Runnable` and passing the `Runnable` to the `post` method of a `View` that is in the main Activity.
private class ViewAdder implements Runnable {
    private View mViewToAdd;

    public ViewAdder(View viewToAdd) {
        mViewToAdd = viewToAdd;
    }

    public void run() {
        mResultsRegion.addView(mViewToAdd);
    }
}

This changes the UI, so could not have been done in the background.

Results
Updating UI with AsyncTask

AsyncTask

• Scenario
  – Total wait time might be large, so you want to show intermediate results
  – You are designing code from the beginning to divide the work between GUI and non-GUI code

• Approach
  – Use AsyncTask with doInBackground & onPostExecute

```java
private class MixedTask extends AsyncTask<...> {
    public SomeType doInBackground(...) {
        doNonGuiStuff();
        return(valueForUiThread);
    }
    public void onPostExecute(SomeType valueForUiThread) {
        doGuiStuffWith(valueForUiThread);
    }
}
```
AsyncTask: Quick Example

• Task itself
  
  ```java
  private class ImageDownloadTask extends AsyncTask<String, Void, View> {
      public View doInBackground(String... urls) {
          return (BitmapUtils.viewForImage(urls[0], MainActivity.this));
      }

      public void onPostExecute(View viewToAdd) {
          mResultsRegion.addView(viewToAdd);
      }
  }
  ```

• Invoking task
  
  ```java
  String imageAddress = "http://...";
  ImageDownloadTask task = new ImageDownloadTask();
  task.execute(imageAddress);
  ```

AsyncTask Details: Constructor

• Class is genericized with three arguments
  
  - AsyncTask<ParamType, ProgressType, ResultType>

• Interpretation
  
  - ParamType
    - This is the type you pass to execute, which in turn is the type that is send to doInBackground. Both methods use varargs, so you can send any number of params.
  
  - ProgressType
    - This is the type that you pass to publishProgress, which in turn is passed to onProgressUpdate (which is called in UI thread). Use Void if you do not need to display intermediate progress.
  
  - ResultType
    - This is the type that you should return from doInBackground, which in turn is passed to onPostExecute (which is called in UI thread).
AsyncTask Details: doInBackground

- **Idea**
  - This is the code that gets executed in the background. It cannot update the UI.
  - It takes as arguments whatever was passed to execute
  - It returns a result that will be later passed to onPostExecute in the UI thread.

- **Code**
  ```java
  private class SomeTask extends AsyncTask<Type1, Void, Type2> {
    public Type2 doInBackground(Type1... params) {
      return(doNonUiStuffWith(params));
    }
  }
  ...
  new SomeTask().execute(type1VarA, type1VarB);
  ```

AsyncTask Details: onPostExecute

- **Idea**
  - This is the code that gets executed by the UI thread. It can update the UI.
  - It takes as an argument whatever was returned by doInBackground

- **Code**
  ```java
  private class SomeTask extends AsyncTask<Type1, Void, Type2> {
    public Type2 doInBackground(Type1... params) {
      return(doNonUiStuffWith(params));
    }
    public void onPostExecute(Type2 result) { doUiStuff(result); }
  }
  ...
  new SomeTask().execute(type1VarA, type1VarB);
  ```
AsyncTask Details: Other Methods

- **onPreExecute**
  - Invoked by the UI thread before doInBackground starts

- **publishProgress**
  - Sends an intermediate update value to onProgressUpdate. You call this from code that is in doInBackground. The type is the middle value of the class declaration.

- **onProgressUpdate**
  - Invoked by the UI thread. Takes as input whatever was passed to publishProgress.

- **Note**
  - All of these methods can be omitted.

Example: Multithreaded Image Viewer (Version 2)

- **Idea**
  - Let user enter URLs of images to be displayed
  - Download and display them inside a scrolling horizontal LinearLayout

- **Approach**
  - Extend previous example. Make an AsyncTask
    - doInBackground
      - Download images and associate them with ImageView, as shown in the previous subsection. If there was an error, create a TextView saying so. Return that View.
    - onPostExecute
      - Add the View to the LinearLayout
public class ImageDownloader2Activity extends ImageDownloader1Activity {
    @Override
    protected void addImages(String[] images) {
        for (String image: images) {
            ImageDownloadTask task = new ImageDownloadTask();
            task.execute(image);
        }
    }

    private class ImageDownloadTask extends AsyncTask<String, Void, View> {
        @Override
        public View doInBackground(String... urls) {
            return(BitmapUtils.viewForImage(urls[0],
                ImageDownloader2Activity.this));
        }

        @Override
        public void onPostExecute(View viewToAdd) {
            mResultsRegion.addView(viewToAdd);
        }
    }
}

Results

Multithreading Examples

URLs:
http://coreservlets.com/images/CSAJSP-Book1Ed.jpg
http://coreservlets.com/images/CSAJSP-Bulgarian.jpg
http://coreservlets.com/images/CSAJSP-Chinese.jpg
http://coreservlets.com/images/CSAJSP-Spanish.jpg

Download Images
Wrap-Up

More Reading

• **JavaDoc**
  – AsyncTask

• **Tutorial: Processes and Threads**

• **Chapters**
  – Threads, Services, Receivers, and Alerts
    • From *The Android Developer’s Cookbook* by Steele & To
  – Dealing with Threads
    • From *The Busy Coder’s Guide to Android Development* by Mark Murphy (http://commonsware.com/Android/)
Summary

• **Update UI after all threads done**
  – One update per group of tasks
    • Use taskList.awaitTermination

• **Update UI incrementally with post**
  – Potentially several updates per task
    • Main thread does non-GUI stuff. Sends values for UI thread via someView.post(someRunnable)

• **Update UI incrementally with AsyncTask**
  – One update per task, but several updates per group of tasks
    • doInBackground
      – Does non-GUI stuff. Returns value for UI thread.
    • doPostExecute
      – Does GUI stuff. Uses value from doInBackground

• **Loading images from network**
  – Use HttpURLConnection & BitmapFactory.decodeStream

Questions?

Customized Java EE Training: http://courses.coreservlets.com/
Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android. Developed and taught by well-known author and developer. At public venues or onsite at your location.