Android Networking
HTTP and Background Tasks

Javed Hasan

BJIT Limited

HTTP and Background Tasks

Introduce PhotoGallery Application

- Fetch photo information from flickr
- Show photo captions in Gallery View

Creating Photo Gallery Apps

- GridView Basics

Android Networking Basics

- Basic networking code
- AsyncTask to Run on a Background Thread

Fetching XML from flickr

- Model GalleryItem Objects
- Use XMLPullParser Interface to get GalleryItem contents from flickr

PhotoGallery App

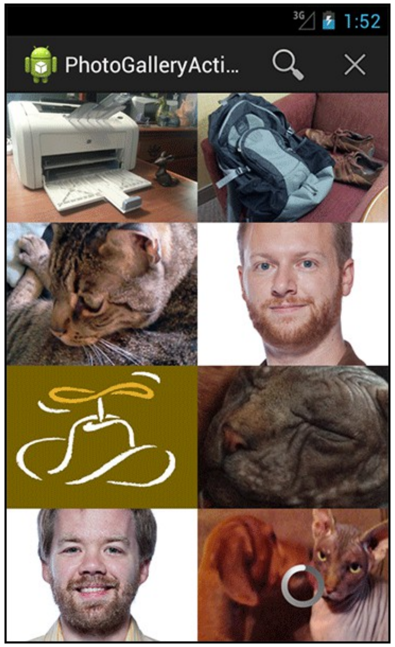
PhotoGallery is a client for the photo-sharing site Flickr.

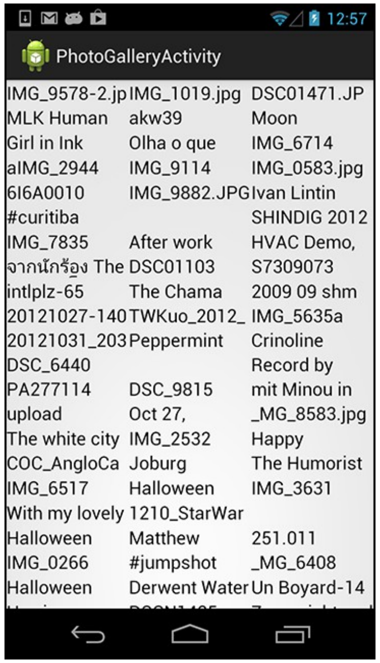
It will fetch and display the most recent public photos uploaded to Flickr.

You will learn how to use Android's high-level HTTP networking.

In this lesson, you will be fetching, parsing, and displaying photo captions from Flickr.

Complete PhotoGallery App





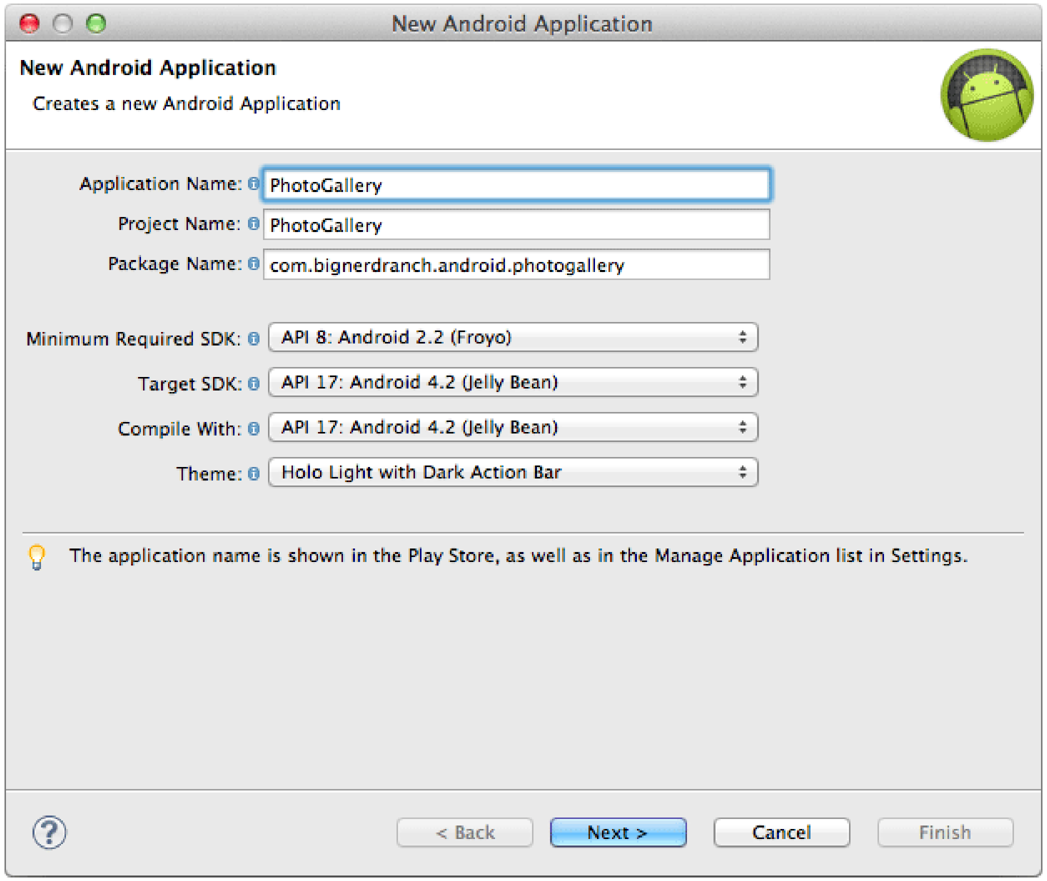
```
IMG_9578-2.jp   IMG_1019.jpg    DSC01471.JP
MLK Human       akw39           Moon
Girl in Ink     Olha o que      IMG_6714
aIMG_2944       IMG_9114        IMG_0583.jpg
6I6A0010        IMG_9882.JPG    Ivan Lintin
#curitiba                       SHINDIG 2012
IMG_7835        After work      HVAC Demo,
จากนักร้อง The   DSC01103        S7309073
intlplz-65      The Chama       2009 09 shm
20121027-140    TWKuo_2012_     IMG_5635a
20121031_203    Peppermint      Crinoline
DSC_6440                        Record by
PA277114        DSC_9815        mit Minou in
upload          Oct 27,         _MG_8583.jpg
The white city  IMG_2532        Happy
COC_AngloCa     Joburg          The Humorist
IMG_6517        Halloween       IMG_3631
With my lovely  1210_StarWar
Halloween       Matthew         251.011
IMG_0266        #jumpshot       _MG_6408
Halloween       Derwent Water   Un Boyard-14
```

Show Caption for this Lesson

Create PhotoGallery Project

Activity setup
(PhotoGalleryActivity.java)

```
~~public class PhotoGalleryActivity extends~~
~~Activity {~~
public class PhotoGalleryActivity extends
 SingleFragmentActivity {
   ~~/* Auto-generated template code */~~
   @Override
   public Fragment createFragment() {
      return new PhotoGalleryFragment();
   }
}
```


New Android Application

New Android Application

Creates a new Android Application

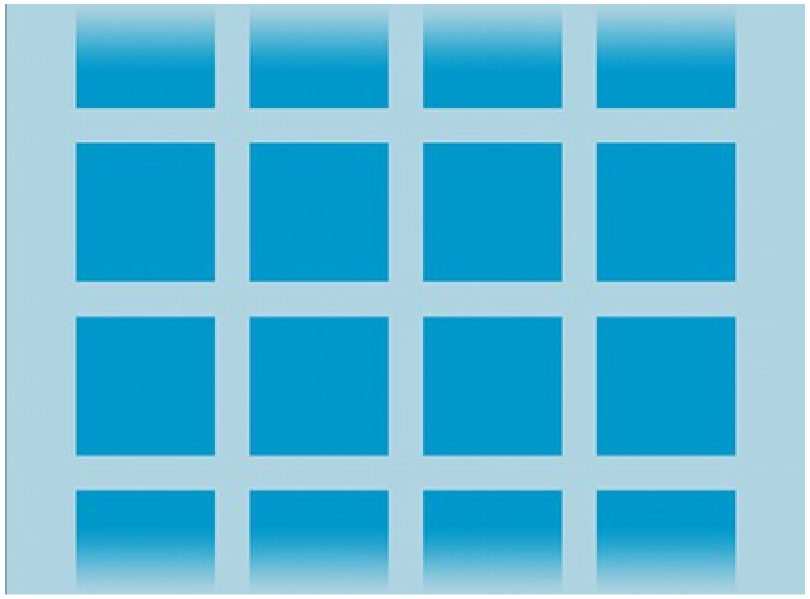
Application Name:	PhotoGallery
Project Name:	PhotoGallery
Package Name:	com.bignerdranch.android.photogallery
Minimum Required SDK:	API 8: Android 2.2 (Froyo)
Target SDK:	API 17: Android 4.2 (Jelly Bean)
Compile With:	API 17: Android 4.2 (Jelly Bean)
Theme:	Holo Light with Dark Action Bar

The application name is shown in the Play Store, as well as in the Manage Application list in Settings.

GridView Basics

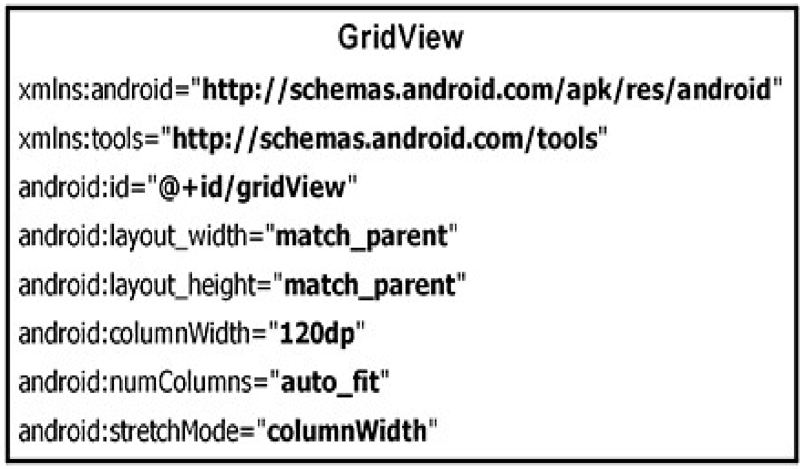
GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

The grid items are automatically inserted to the layout using a **ListAdapter**.



Make Photo Gallery Fragment Layout

A GridView
(layout/fragment_photo_gallery.xml)



```
GridView
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:id="@+id/gridView"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:columnWidth="120dp"
android:numColumns="auto_fit"
android:stretchMode="columnWidth"
```

the stretchMode attribute tells the **GridView** to divide the extra space equally among the columns.

Some skeleton code (PhotoGalleryFragment.java)

```
package com.bignerdranch.android.photogallery;

...
public class PhotoGalleryFragment extends Fragment {
   GridView mGridView;
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setRetainInstance(true);
   }
@Override
   public View onCreateView(LayoutInflater inflater,
         ViewGroup container, Bundle savedInstanceState) {

         View v = inflater.inflate(R.layout.fragment_photo_gallery,
            container, false);
         mGridView = (GridView)v.findViewById(R.id.gridView);
         return v;
   }
}
```

Android Networking Basics

(FlickrFetchr.java)

```
public class FlickrFetchr {
   byte[] getUrlBytes(String urlSpec) throws IOException {
      URL url = new URL(urlSpec);
      HttpURLConnection connection = (HttpURLConnection)url.openConnection();
      try {
         ByteArrayOutputStream out = new ByteArrayOutputStream();
         InputStream in = connection.getInputStream();
         if (connection.getResponseCode() != HttpURLConnection.HTTP_OK) {
            return null;
         }
         int bytesRead = 0;
         byte[] buffer = new byte[1024];
         while ((bytesRead = in.read(buffer)) > 0) {
            out.write(buffer, 0, bytesRead);
         }
         out.close();
         return out.toByteArray();
      } finally {
                  connection.disconnect();
      }
   }

public String getUrl(String urlSpec) throws IOException
{
      return new String(getUrlBytes(urlSpec));
 }
}
```

FlickrFetchr class will handle the networking in **PhotoGallery**.

getUrlBytes(String) method fetches raw data from a URL and returns it as an array of bytes.

The getUrl(String) method converts the result from getUrlBytes(String) to a String.

Asking Permission to Network

Add networking permission to manifest (AndroidManifest.xml)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.bignerdranch.android.photogallery"
  android:versionCode="1"
  android:versionName="1.0" >
  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="15" />
  <uses-permission android:name="android.permission.INTERNET" />
  ...
</manifest>
```

Use AsyncTask to Run a Background Thread 1/2

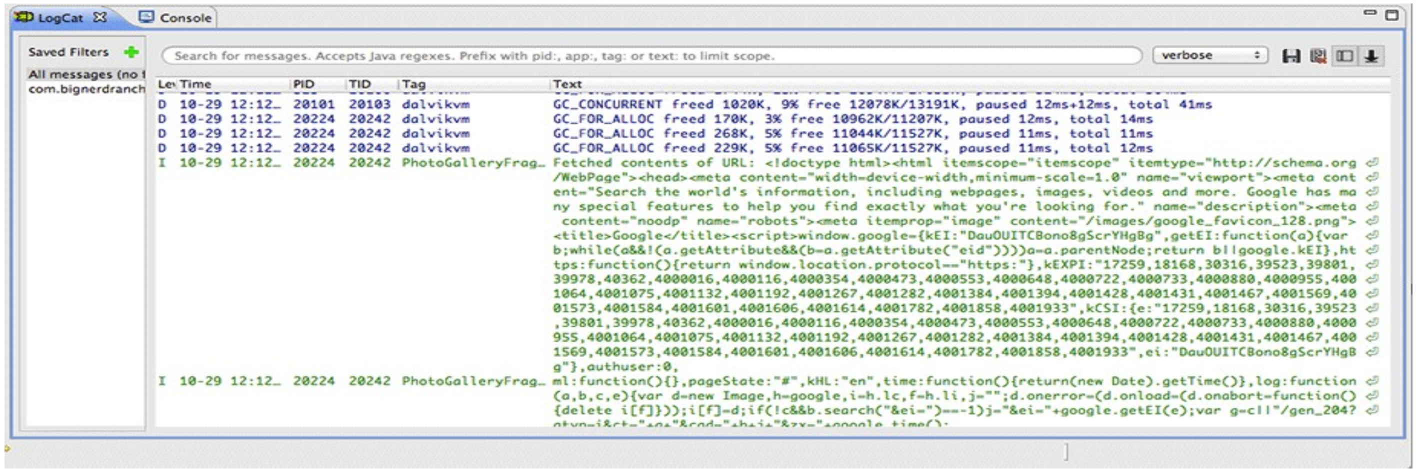
You cannot simply call FlickrFetchr.getURL(String) directly in **PhotoGalleryFragment**. Instead, you need to create a background thread and run your code there. **AsyncTask** creates a background thread for you and runs the code in the doInBackground(…) method on that thread.

```
public class PhotoGalleryFragment extends Fragment {
   private static final String TAG = "PhotoGalleryFragment";
   GridView mGridView;
   ...
   private class FetchItemsTask extends AsyncTask<Void,Void,Void> {
      @Override
      protected Void doInBackground(Void... params) {
         try {
            String result = new FlickrFetchr().getUrl("http://www.google.com");
            Log.i(TAG, "Fetched contents of URL: " + result);
         } catch (IOException ioe) {
            Log.e(TAG, "Failed to fetch URL: ", ioe);
         }
         return null;
      }
   }
}
```

Use AsyncTask to Run a Background Thread 2/2

In PhotoGalleryFragment.onCreate(…), call execute() on a new instance of FetchItemsTask. The call to execute() will start your AsyncTask, which will then fire up its background thread and call doInBackground(…).

```
public class PhotoGalleryFragment extends Fragment {
    private static final String TAG = "PhotoGalleryFragment";
    GridView mGridView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        new FetchItemsTask().execute();
    }
    ...
}
```



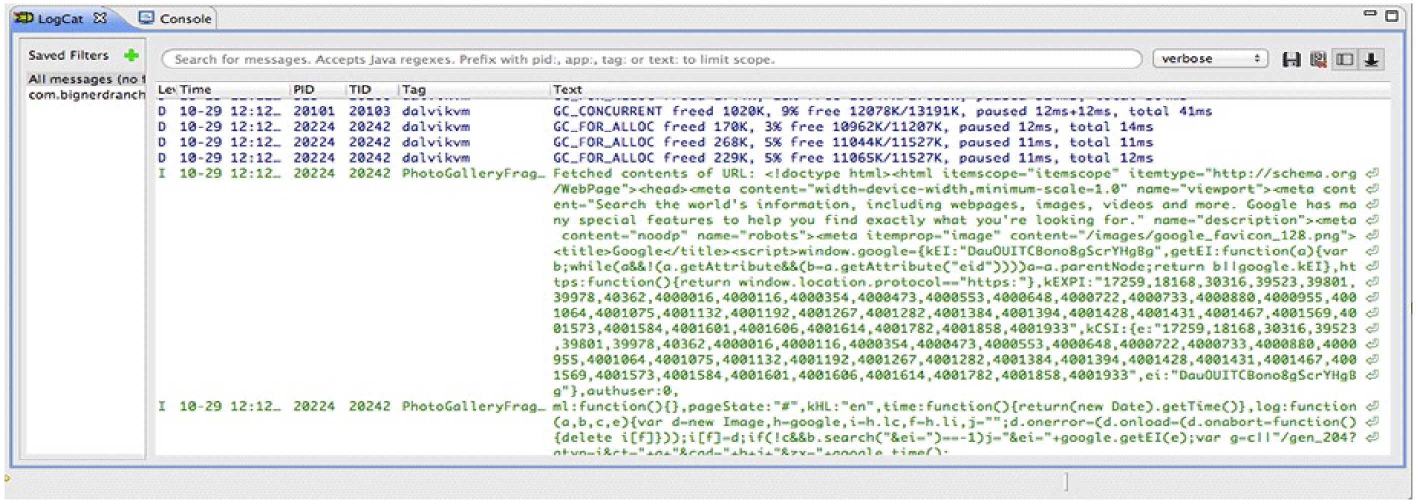
```
Lev Time          PID    TID    Tag                  Text
D  10-29 12:12…  20101  20103  dalvikvm             GC_CONCURRENT freed 1020K, 9% free 12078K/13191K, paused 12ms+12ms, total 41ms
D  10-29 12:12…  20224  20242  dalvikvm             GC_FOR_ALLOC freed 170K, 3% free 10962K/11207K, paused 12ms, total 14ms
D  10-29 12:12…  20224  20242  dalvikvm             GC_FOR_ALLOC freed 268K, 5% free 11044K/11527K, paused 11ms, total 11ms
D  10-29 12:12…  20224  20242  dalvikvm             GC_FOR_ALLOC freed 229K, 5% free 11065K/11527K, paused 11ms, total 12ms
I  10-29 12:12…  20224  20242  PhotoGalleryFrag…    Fetched contents of URL: <!doctype html><html itemscope="itemscope" itemtype="http://schema.org
                                                     /WebPage"><head><meta content="width=device-width,minimum-scale=1.0" name="viewport"><meta cont
                                                     ent="Search the world's information, including webpages, images, videos and more. Google has ma
                                                     ny special features to help you find exactly what you're looking for." name="description"><meta
                                                      content="noodp" name="robots"><meta itemprop="image" content="/images/google_favicon_128.png">
                                                     <title>Google</title><script>window.google={kEI:"DauOUITCBono8gScrYHgBg",getEI:function(a){var
                                                     b;while(a&&!(a.getAttribute&&(b=a.getAttribute("eid"))))a=a.parentNode;return b||google.kEI},ht
                                                     tps:function(){return window.location.protocol=="https:"},kEXPI:"17259,18168,30316,39523,39801,
                                                     39978,40362,4000016,4000116,4000354,4000473,4000553,4000648,4000722,4000733,4000880,4000955,400
                                                     1064,4001075,4001132,4001192,4001267,4001282,4001384,4001394,4001428,4001431,4001467,4001569,40
                                                     01573,4001584,4001601,4001606,4001614,4001782,4001858,4001933",kCSI:{e:"17259,18168,30316,39523
                                                     ,39801,39978,40362,4000016,4000116,4000354,4000473,4000553,4000648,4000722,4000733,4000880,4000
                                                     955,4001064,4001075,4001132,4001192,4001267,4001282,4001384,4001394,4001428,4001431,4001467,400
                                                     1569,4001573,4001584,4001601,4001606,4001614,4001782,4001858,4001933",ei:"DauOUITCBono8gScrYHgB
                                                     g"},authuser:0,
I  10-29 12:12…  20224  20242  PhotoGalleryFrag…    ml:function(){},pageState:"#",kHL:"en",time:function(){return(new Date).getTime()},log:function
                                                     (a,b,c,e){var d=new Image,h=google,i=h.lc,f=h.li,j="";d.onerror=(d.onload=(d.onabort=function()
                                                     {delete i[f]}));i[f]=d;if(!c&&b.search("&ei=")==-1)j="&ei="+google.getEI(e);var g=c||"/gen_204?
```

LogCat
Console
Saved Filters
All messages (no f
com.bignerdranch
Search for messages. Accepts Java regexes. Prefix with pid:, app:, tag: or text: to limit scope.
verbose

Use AsyncTask to Run a Background Thread 2/2

In PhotoGalleryFragment.onCreate(…), call execute() on a new instance of FetchItemsTask. The call to execute() will start your AsyncTask, which will then fire up its background thread and call doInBackground(…).

```
public class PhotoGalleryFragment extends Fragment {
    private static final String TAG = "PhotoGalleryFragment";
    GridView mGridView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        new FetchItemsTask().execute();
    }
    ...
}
```


LogCat
Console
Saved Filters
All messages (no f
com.bignerdranch
Search for messages. Accepts Java regexes. Prefix with pid:, app:, tag: or text: to limit scope.
verbose

```
Le Time          PID    TID    Tag                 Text
D  10-29 12:12…  20101  20103  dalvikvm            GC_CONCURRENT freed 1020K, 9% free 12078K/13191K, paused 12ms+12ms, total 41ms
D  10-29 12:12…  20224  20242  dalvikvm            GC_FOR_ALLOC freed 170K, 3% free 10962K/11207K, paused 12ms, total 14ms
D  10-29 12:12…  20224  20242  dalvikvm            GC_FOR_ALLOC freed 268K, 5% free 11044K/11527K, paused 11ms, total 11ms
D  10-29 12:12…  20224  20242  dalvikvm            GC_FOR_ALLOC freed 229K, 5% free 11065K/11527K, paused 11ms, total 12ms
I  10-29 12:12…  20224  20242  PhotoGalleryFrag…   Fetched contents of URL: <!doctype html><html itemscope="itemscope" itemtype="http://schema.org
                                                   /WebPage"><head><meta content="width=device-width,minimum-scale=1.0" name="viewport"><meta cont
                                                   ent="Search the world's information, including webpages, images, videos and more. Google has ma
                                                   ny special features to help you find exactly what you're looking for." name="description"><meta
                                                    content="noodp" name="robots"><meta itemprop="image" content="/images/google_favicon_128.png">
                                                   <title>Google</title><script>window.google={kEI:"DauOUITCBono8gScrYHgBg",getEI:function(a){var
                                                   b;while(a&&!(a.getAttribute&&(b=a.getAttribute("eid"))))a=a.parentNode;return b||google.kEI},ht
                                                   tps:function(){return window.location.protocol=="https:"},kEXPI:"17259,18168,30316,39523,39801,
                                                   39978,40362,4000016,4000116,4000354,4000473,4000553,4000648,4000722,4000733,4000880,4000955,400
                                                   1064,4001075,4001132,4001192,4001267,4001282,4001384,4001394,4001428,4001431,4001467,4001569,40
                                                   01573,4001584,4001601,4001606,4001614,4001782,4001858,4001933",kCSI:{e:"17259,18168,30316,39523
                                                   ,39801,39978,40362,4000016,4000116,4000354,4000473,4000553,4000648,4000722,4000733,4000880,4000
                                                   955,4001064,4001075,4001132,4001192,4001267,4001282,4001384,4001394,4001428,4001431,4001467,400
                                                   1569,4001573,4001584,4001601,4001606,4001614,4001782,4001858,4001933",ei:"DauOUITCBono8gScrYHgB
                                                   g"},authuser:0,
I  10-29 12:12…  20224  20242  PhotoGalleryFrag…   ml:function(){},pageState:"#",kHL:"en",time:function(){return(new Date).getTime()},log:function
                                                   (a,b,c,e){var d=new Image,h=google,i=h.lc,f=h.li,j="";d.onerror=(d.onload=(d.onabort=function()
                                                   {delete i[f]}));i[f]=d;if(!c&&b.search("&ei=")==-1)j="&ei="+google.getEI(e);var g=c||"/gen_204?
```

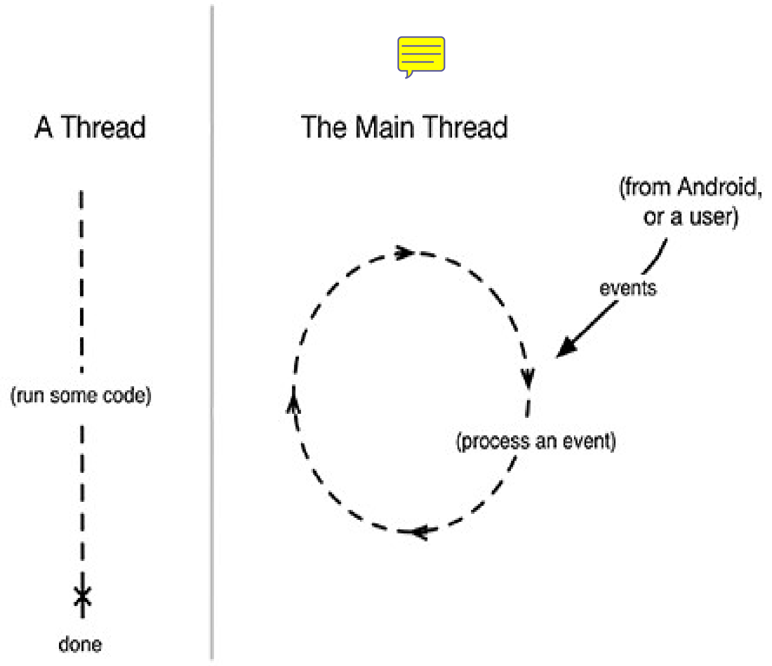
Threads in Android 1/2

A thread is a single sequence of execution. Code running within a single thread will execute one step after another.

Every Android app starts life with a main thread. The main thread, however, isn't a preordained list of steps.

Instead, it sits in an infinite loop and waits for events initiated by the user or the system.

Then it executes code in response to those events as they occur.

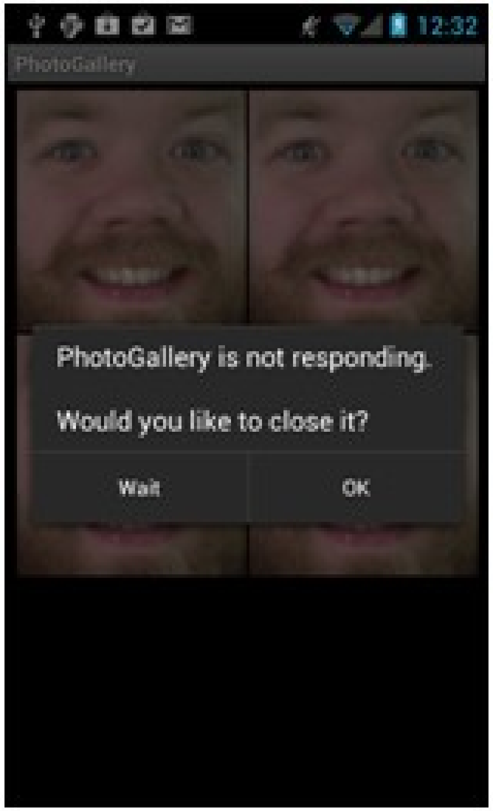


Threads in Android 2/2

Networking takes a long time compared to other tasks. During that time, the user interface will be completely unresponsive, which might result in an application not responding, or ANR.

An ANR occurs when Android's watchdog determines that the main thread has failed to respond to an important event, like pressing the back button.

This is why Android disallowed network operations on the main thread starting with Honeycomb. So, you create a background thread and access the network from there.



PhotoGallery is not responding.

Would you like to close it?

Fetching XML From Flickr 1/4

Flickr offers a fine XML API. Details at: www.flickr.com/services/api/. You will be using REST. The API endpoint is http://api.flickr.com/services/rest/. You can invoke the methods Flickr provides on this endpoint.

In the API documentation, locate flickr.photos.getRecent. This method "Returns a list of the latest public photos uploaded to flickr." That is exactly what you need for PhotoGallery.

The only required parameter for the getRecent method is an API key. To get an API key, return to http://www.flickr.com/services/api/ and follow the link for API keys.

Fetching XML From Flickr 2/4

Time to start coding. First, add some constants to FlickrFetchr.

```
public class FlickrFetchr {
    public static final String TAG = "FlickrFetchr";
    private static final String ENDPOINT = "http://api.flickr.com/services/rest/";
    private static final String API_KEY = "yourApiKeyHere";
    private static final String METHOD_GET_RECENT = "flickr.photos.getRecent";
    private static final String PARAM_EXTRAS = "extras";
    private static final String EXTRA_SMALL_URL = "url_s";
```

These constants define the endpoint, the method name, the API key, and one extra parameter called extras, with a value of url_s.

Specifying the url_s extra tells Flickr to include the URL for the small version of the picture if it is available.

Now use the constants to write a method that builds an appropriate request URL and fetches its contents.

Fetching XML From Flickr 3/4

Add fetchItems() method (FlickrFetchr.java)

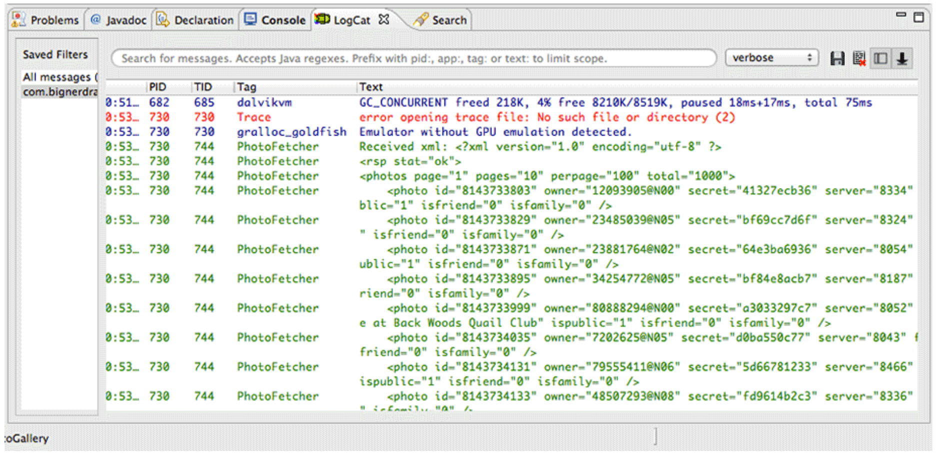
```
public class FlickrFetchr {
    ...
    String getUrl(String urlSpec) throws IOException {
        return new String(getUrlBytes(urlSpec));
    }
    public void fetchItems() {
        try {
            String url = Uri.parse(ENDPOINT).buildUpon()
                    .appendQueryParameter("method", METHOD_GET_RECENT)
                    .appendQueryParameter("api_key", API_KEY)
                    .appendQueryParameter(PARAM_EXTRAS, EXTRA_SMALL_URL)
                    .build().toString();
            String xmlString = getUrl(url);
            Log.i(TAG, "Received xml: " + xmlString);
        } catch (IOException ioe) {
            Log.e(TAG, "Failed to fetch items", ioe);
        }
    }
}
```

Fetching XML From Flickr 4/4

Call fetchItems() (PhotoGalleryFragment.java)

```
private class FetchItemsTask extends AsyncTask<Void,Void,Void> {
  @Override
  protected Void doInBackground(Void... params) {
     ~~try {~~
        ~~String result = new FlickrFetchr().getUrl("http://www.google.com");~~
       ~~Log.i(TAG, "Fetched contents of URL: " + result);~~
     ~~} catch (IOException ioe) {~~
        ~~Log.e(TAG, "Failed to fetch URL: ", ioe);~~
     ~~}~~
     new FlickrFetchr().fetchItems();
     return null;
  }
}
```

Run PhotoGallery and you should see rich, fertile Flickr XML in LogCat.



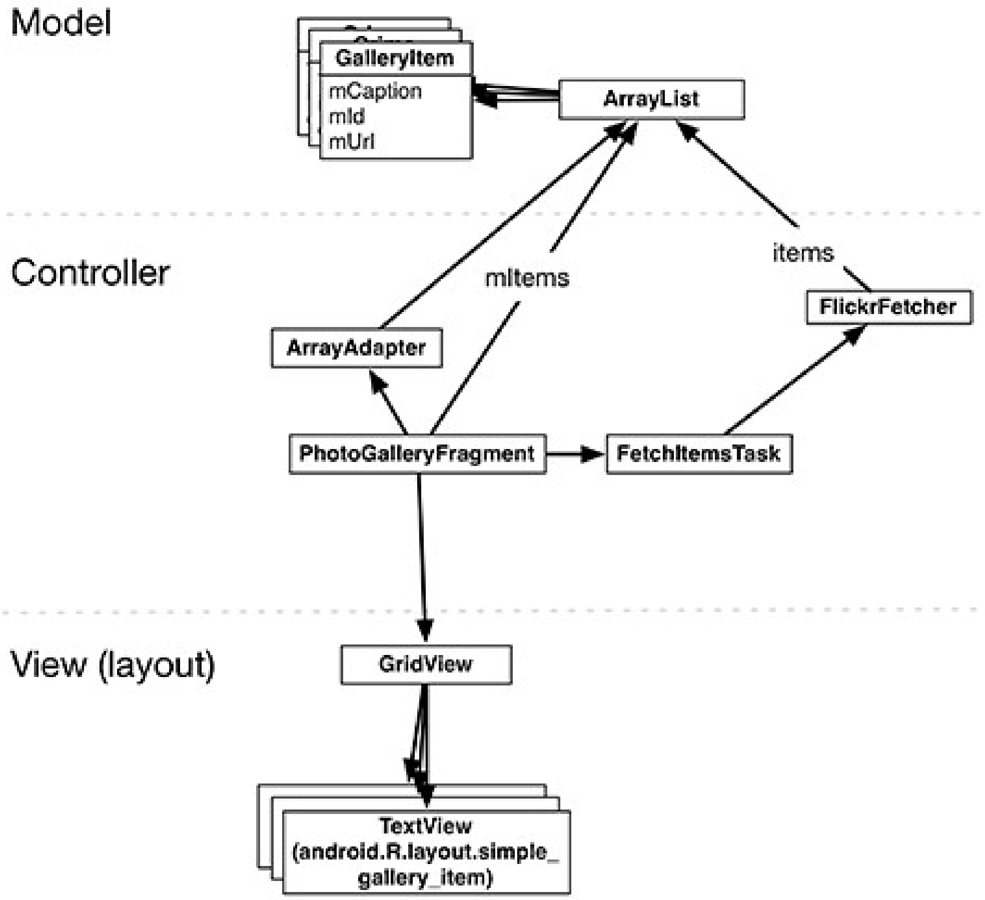
```
        PID   TID   Tag              Text
0:51…   682   685   dalvikvm         GC_CONCURRENT freed 218K, 4% free 8210K/8519K, paused 18ms+17ms, total 75ms
0:53…   730   730   Trace            error opening trace file: No such file or directory (2)
0:53…   730   730   gralloc_goldfish Emulator without GPU emulation detected.
0:53…   730   744   PhotoFetcher     Received xml: <?xml version="1.0" encoding="utf-8" ?>
0:53…   730   744   PhotoFetcher     <rsp stat="ok">
0:53…   730   744   PhotoFetcher     <photos page="1" pages="10" perpage="100" total="1000">
0:53…   730   744   PhotoFetcher         <photo id="8143733803" owner="12093905@N00" secret="41327ecb36" server="8334"
                                     blic="1" isfriend="0" isfamily="0" />
0:53…   730   744   PhotoFetcher         <photo id="8143733829" owner="23485039@N05" secret="bf69cc7d6f" server="8324"
                                     " isfriend="0" isfamily="0" />
0:53…   730   744   PhotoFetcher         <photo id="8143733871" owner="23881764@N02" secret="64e3ba6936" server="8054"
                                     ublic="1" isfriend="0" isfamily="0" />
0:53…   730   744   PhotoFetcher         <photo id="8143733895" owner="34254772@N05" secret="bf84e8acb7" server="8187"
                                     riend="0" isfamily="0" />
0:53…   730   744   PhotoFetcher         <photo id="8143733999" owner="80888294@N00" secret="a3033297c7" server="8052"
                                     e at Back Woods Quail Club" ispublic="1" isfriend="0" isfamily="0" />
0:53…   730   744   PhotoFetcher         <photo id="8143734035" owner="7202625@N05" secret="d0ba550c77" server="8043" f
                                     friend="0" isfamily="0" />
0:53…   730   744   PhotoFetcher         <photo id="8143734131" owner="79555411@N06" secret="5d66781233" server="8466"
                                     ispublic="1" isfriend="0" isfamily="0" />
0:53…   730   744   PhotoFetcher         <photo id="8143734133" owner="48507293@N08" secret="fd9614b2c3" server="8336"
```

Create Model Class GalleryItem

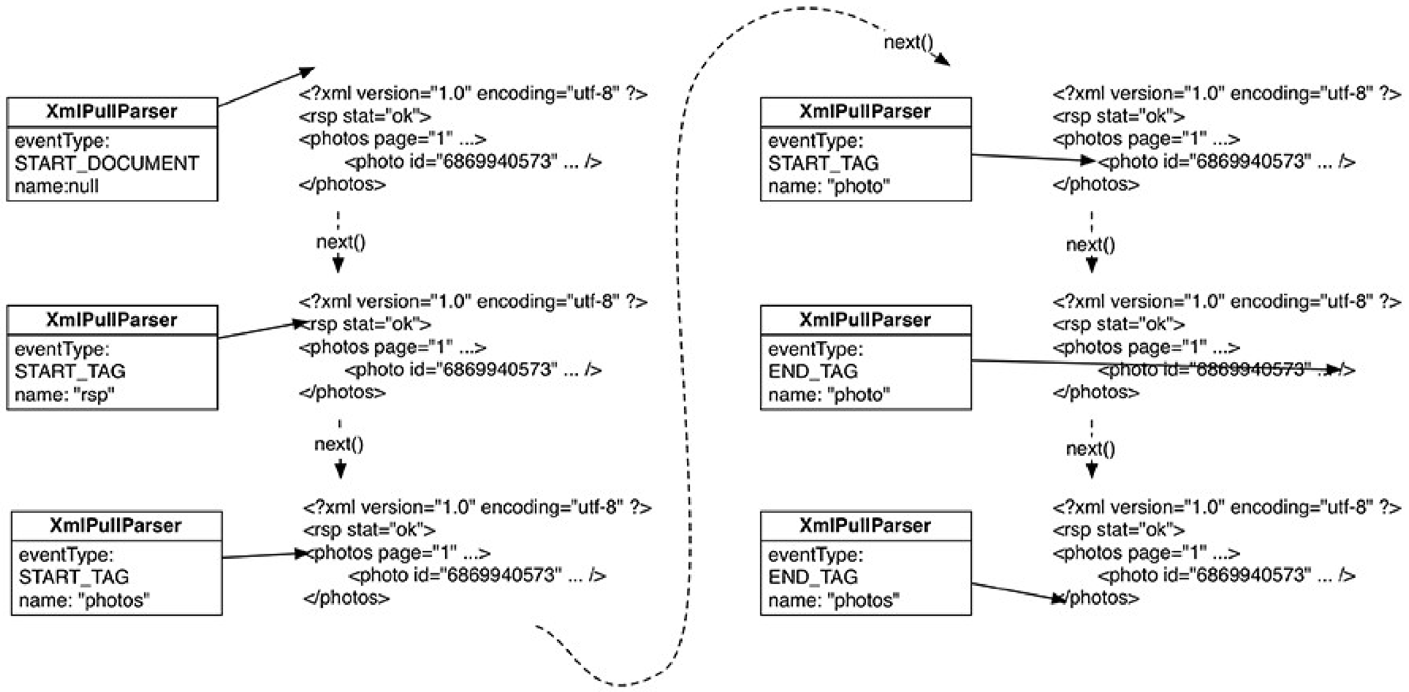
Create model object class (GalleryItem.java)

```
public class GalleryItem {
   private String mCaption;
   private String mId;
   private String mUrl;
   public String toString() {
      return mCaption;
  }
}
```

Have Eclipse generate getters and setters for mId, mCaption, and mUrl.



Use XMLPullParser to Get Photo Info 1/3


XmlPullParser
eventType: START_DOCUMENT name:null
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photos page="1" ...>
<photo id="6869940573" ... />
</photos>
next()
XmlPullParser
eventType: START_TAG name: "rsp"
next()
XmlPullParser
eventType: START_TAG name: "photos"
next()
XmlPullParser
eventType: START_TAG name: "photo"
next()
XmlPullParser
eventType: END_TAG name: "photo"
next()
XmlPullParser
eventType: END_TAG name: "photos"

You can imagine XmlPullParser as having its finger on your XML document, walking step by step through different events like START_TAG, END_TAG, and END_DOCUMENT. At each step, you can call methods like **getText(), getName(), or getAttributeValue(…)** to answer any questions you have about the event XmlPullParser currently has its finger on. To move the finger to the next interesting event in the XML, call **next().** Conveniently, this method also returns the type of event it just moved to.

Use XMLPullParser to Get Photo Info 2/3

Parse Flickr photos (FlickrFetchr.java)

```
public class FlickrFetchr {
    public static final String TAG = "FlickrFetchr";
    private static final String ENDPOINT = "http://api.flickr.com/services/rest/";
    private static final String API_KEY = "your API key";
    private static final String METHOD_GET_RECENT = "flickr.photos.getRecent";
    private static final String XML_PHOTO = "photo";
    ...
    public void fetchItems() {
        ...
    }
    void parseItems(ArrayList<GalleryItem> items, XmlPullParser parser)
            throws XmlPullParserException, IOException {
        int eventType = parser.next();
        while (eventType != XmlPullParser.END_DOCUMENT) {
            if (eventType == XmlPullParser.START_TAG &&
                XML_PHOTO.equals(parser.getName())) {
                String id = parser.getAttributeValue(null, "id");
                String caption = parser.getAttributeValue(null, "title");
                String smallUrl = parser.getAttributeValue(null, EXTRA_SMALL_URL);
GalleryItem item = new GalleryItem();
                item.setId(id);
                item.setCaption(caption);
                item.setUrl(smallUrl);
                items.add(item);
            }
            eventType = parser.next();
        }
    }
}
```

Use XMLPullParser to Get Photo Info 3/3

Call parseItems(...) (FlickrFetchr.java)

```
~~public void fetchItems() {~~
public ArrayList<GalleryItem> fetchItems() {
   ArrayList<GalleryItem> items = new ArrayList<GalleryItem>();
   try {
      String url = Uri.parse(ENDPOINT).buildUpon()
            .appendQueryParameter("method", METHOD_GET_RECENT)
            .appendQueryParameter("api_key", API_KEY)
            .appendQueryParameter(PARAM_EXTRAS, EXTRA_SMALL_URL)
            .build().toString();
      String xmlString = getUrl(url);
      Log.i(TAG, "Received xml: " + xmlString);
      XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
      XmlPullParser parser = factory.newPullParser();
      parser.setInput(new StringReader(xmlString));
      parseItems(items, parser);
   } catch (IOException ioe) {
      Log.e(TAG, "Failed to fetch items", ioe);
   } catch (XmlPullParserException xppe) {
      Log.e(TAG, "Failed to parse items", xppe);
   }
   return items;
}
```

Run PhotoGallery to test your XML parsing code.

PhotoGallery has no way of reporting the contents of your ArrayList right now, so you will need to set a breakpoint and use the debugger if you want to make sure everything worked correctly

From AsyncTask Back to the Main Thread

Implement setupAdapter() (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment extends Fragment {
   private static final String TAG = "PhotoGalleryFragment";
   GridView mGridView;
   ArrayList<GalleryItem> mItems;
   ...
   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
         Bundle savedInstanceState) {
      View v = inflater.inflate(R.layout.fragment_photo_gallery, container, false);
      mGridView = (GridView)v.findViewById(R.id.gridView);
      setupAdapter();
      return v;
   }
   void setupAdapter() {
      if (getActivity() == null || mGridView == null) return;
      if (mItems != null) {
         mGridView.setAdapter(new ArrayAdapter<GalleryItem>
                (getActivity(), android.R.layout.simple_gallery_item, mItems));
      } else {
         mGridView.setAdapter(null);
      }
   }
```

Let's return to the view layer and get PhotoGalleryFragment's GridView to display some captions.

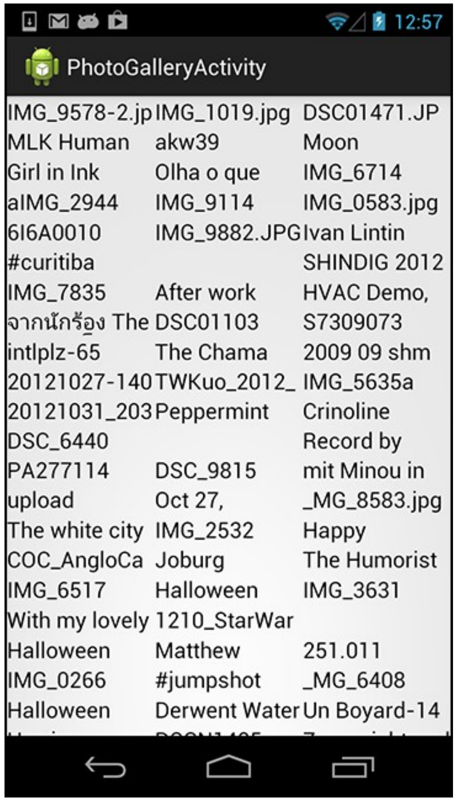
GridView is an AdapterView, like ListView, so it needs an adapter to feed it views to display.

In **PhotoGalleryFragment.java,** add an ArrayList of GalleryItems and then set up an ArrayAdapter that uses a simple layout provided by Android.

Show the Photo Caption

Add adapter update code (PhotoGalleryFragment.java)

```
~~private class FetchItemsTask extends AsyncTask<Void,Void,Void> {~~
private class FetchItemsTask extends AsyncTask<Void,Void,ArrayList<GalleryItem>> {
   @Override
   ~~protected Void doInBackground(Void...  params) {~~
   protected ArrayList<GalleryItem>  doInBackground(Void...  params) {
      ~~new FlickrFetchr().fetchItems();~~
      return new FlickrFetchr().fetchItems();
      ~~return null;~~
   }
   @Override
   protected void onPostExecute(ArrayList<GalleryItem>  items) {
      mItems = items;
      setupAdapter();
   }
}
```



IMG_9578-2.jp	IMG_1019.jpg	DSC01471.JP
MLK Human	akw39	Moon
Girl in Ink	Olha o que	IMG_6714
aIMG_2944	IMG_9114	IMG_0583.jpg
6I6A0010	IMG_9882.JPG	Ivan Lintin
#curitiba		SHINDIG 2012
IMG_7835	After work	HVAC Demo,
จากนักร้อง The	DSC01103	S7309073
intlplz-65	The Chama	2009 09 shm
20121027-140	TWKuo_2012_	IMG_5635a
20121031_203	Peppermint	Crinoline
DSC_6440		Record by
PA277114	DSC_9815	mit Minou in
upload	Oct 27,	_MG_8583.jpg
The white city	IMG_2532	Happy
COC_AngloCa	Joburg	The Humorist
IMG_6517	Halloween	IMG_3631
With my lovely	1210_StarWar	
Halloween	Matthew	251.011
IMG_0266	#jumpshot	_MG_6408
Halloween	Derwent Water	Un Boyard-14

AsyncTask has another method you can override called onPostExecute(…). onPostExecute(…) is run after doInBackground(…) completes. In addition, onPostExecute(…) is run on the main thread, not the background thread, so it is safe to update the UI within it.

Challenge

By default, **getRecent** returns one page of 100 results. There is an additional parameter you can use called **page** that will let you return page two, three, and so on.

For this challenge, add code to your adapter that detects when you are at the end of your array of items and replaces the current page with the next page of results.

Android Networking
Loopers, Handlers, and HandlerThread

Javed Hasan
BJIT Limited

Loopers, Handlers, and HandlerThread

PhotoGallery Application New Feature

- Download and Show flickr photos
- Show photo in Gallery View

Downloading Lots of Small Things

- Using a background thread to download
- Communicating with Main Thread
- Assembling a background thread

Messages and Message Handlers

- Message Anatomy, Handler Anatomy
- Using Handlers and Passing Handlers

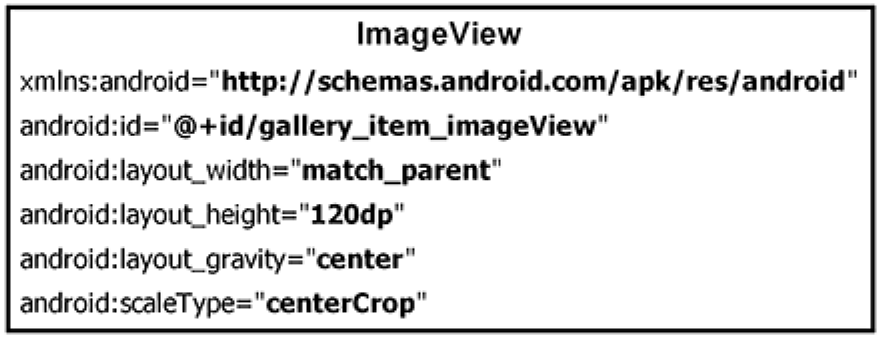
PhotoGallery App New Feature

You will learn how to use **Looper**, **Handler**, and **HandlerThread** to dynamically download and display photos in PhotoGallery.

Preparing GridView for Displaying Images 1/2

To display photos, you need a custom adapter that provides **ImageViews**. Each ImageView will display a photo downloaded from the mUrl of a GalleryItem.

Gallery item layout (res/layout/gallery_item.xml)



ImageView

```
xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/gallery_item_imageView"
android:layout_width="match_parent"
android:layout_height="120dp"
android:layout_gravity="center"
android:scaleType="centerCrop"
```

To make the most of the ImageView's space, you have set its scaleType to centerCrop. This setting centers the image and then scales it up so that the smaller dimension is equal to the view and the larger one is cropped on both sides.

Preparing GridView for Displaying Images 2/2

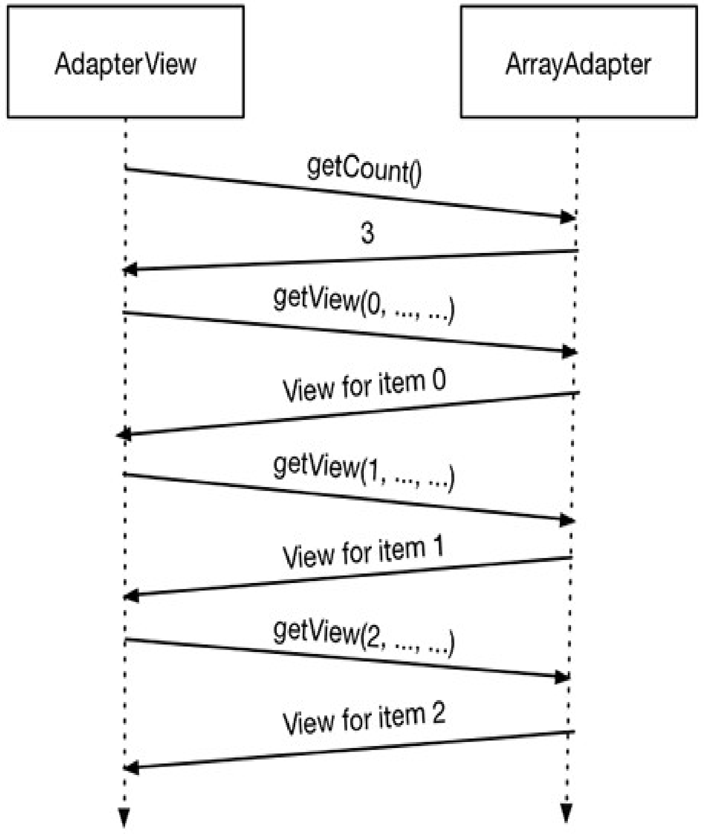
Create GalleryItemAdapter (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment  extends Fragment {
   ...
   void setupAdapter() {
      if (getActivity() == null || mGridView == null) return;
      if (mItems != null) {
         ~~mGridView.setAdapter(~~
              ~~new ArrayAdapter<GalleryItem>  (getActivity(),~~
              ~~android.R.layout.simple_gallery_item,  mItems));~~
         mGridView.setAdapter(
              new GalleryItemAdapter(mItems));
      } else {
         mGridView.setAdapter(null);
      }
   }
```

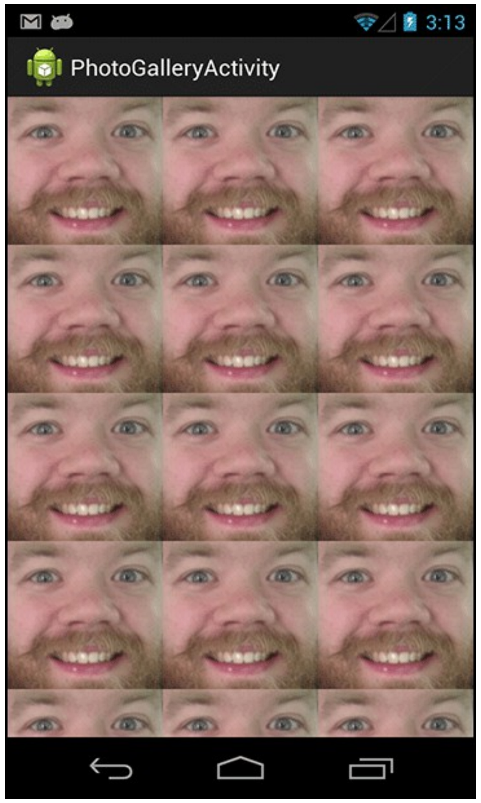
```
private class FetchItemsTask extends AsyncTask<Void,
       Void,ArrayList<GalleryItem>> {
      ...
   }
   private class GalleryItemAdapter extends ArrayAdapter<GalleryItem> {
      public GalleryItemAdapter(ArrayList<GalleryItem> items) {
         super(getActivity(), 0, items);
      }
      @Override
      public View getView(int position, View convertView, ViewGroup parent) {
         if (convertView == null) {
            convertView = getActivity().getLayoutInflater()
                  .inflate(R.layout.gallery_item, parent, false);
         }
         ImageView imageView = (ImageView)convertView
               .findViewById(R.id.gallery_item_imageView);
         imageView.setImageResource(R.drawable.brian_up_close);
         return convertView;
      }
   }
}
```

Preparing GridView for Displaying Images 2/2

AdapterView-ArrayAdapter pong



Wall of Images from Resource Folder



Downloading Lots of Small Things

One way is to download images all at once. But this option has 2 problems:

1. Downloading could take a while, and the UI would not be updated until the downloading was complete.
2. The cost of having to store the entire set of images.

Real world apps often choose to download images only when they need to be displayed on screen. Downloading on demand puts the responsibility on the **GridView** and its adapter. The adapter will trigger the image downloading as part of its **getView(…)** implementation.

AsyncTask is the easiest way to get a background thread, but it is fundamentally ill-suited for repetitive and long-running work.

Instead of using an **AsyncTask**, you are going to create a dedicated background thread. This is the most common way to implement downloading on an as-needed basis.

Communicating with the Main Thread

Your dedicated thread will download photos, but how will it work with the **GridView's** adapter to display them when it cannot directly access the main thread?

Each thread will have an Inbox to communicate with the other thread. For instance, Background thread writes a message about the image gets downloaded and puts it on top of Main thread's inbox. Main thread does the same thing when he wants to tell Background thread that the download the thumbnail at specific URL.

In Android, the inbox that threads use is called a *message queue*. A thread that works by using a *message queue* is called a *message loop*; it loops again and again looking for new messages on its queue

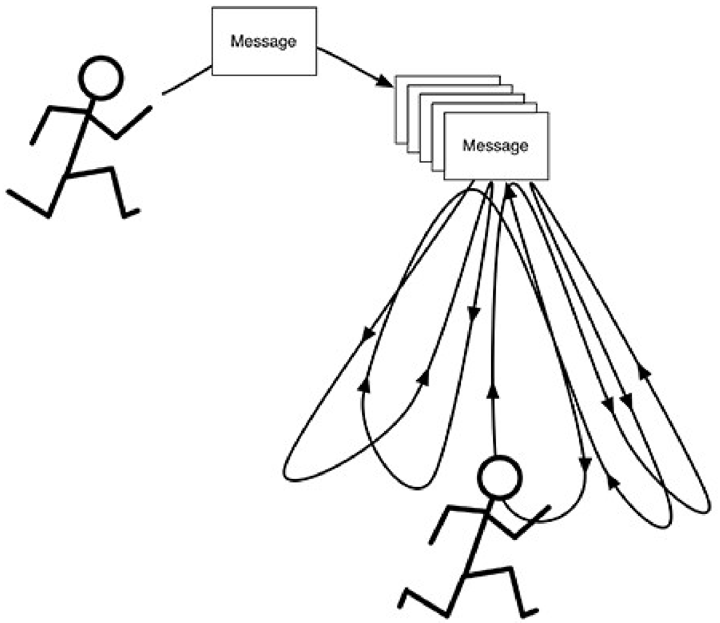
Communicating with the Main Thread

A message loop consists of a thread and a looper. The **Looper** is the object that manages a thread's message queue.

The main thread is a message loop and has a looper. Everything your main thread does is performed by its looper, which grabs messages off of its message queue and performs the task it specifies.

You are going to create a background thread that is a message loop, too. You will use a class called **HandlerThread** that prepares a **Looper** for you.

Looper Scan for Messages



Assembling a Background Thread 1/3

Initial thread code (ThumbnailDownloader.java)

```
public class ThumbnailDownloader<Token> extends HandlerThread {
   private static final String TAG = "ThumbnailDownloader";

   public ThumbnailDownloader() {
      super(TAG);
   }
   public void queueThumbnail(Token token, String url) {
      Log.i(TAG, "Got an URL: " + url);
   }
}
```

Your **ThumbnailDownloader's** user will need to use some object to identify each download, so give it one generic argument called Token by naming it **ThumbnailDownloader<Token>**.

queueThumbnail() expects a Token and a String. This is the method you will have **GalleryItemAdapter** call in its **getView(…)** implementation.

Assembling a Background Thread 2/3

Create ThumbnailDownloader (PhotoGalleryFragment.java)

```
public class PhotoGalleryFragment extends Fragment {
    private static final String TAG = "PhotoGalleryFragment";
    GridView mGridView;
    ArrayList<GalleryItem> mItems;
    ThumbnailDownloader<ImageView> mThumbnailThread;
     @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        new FetchItemsTask().execute();
        mThumbnailThread =
                new ThumbnailDownloader<ImageView>();
        mThumbnailThread.start();
        mThumbnailThread.getLooper();
        Log.i(TAG, "Background thread started");
    }
```

```
@Override
    public View onCreateView(LayoutInflater inflater, ViewGr
oup container,
            Bundle savedInstanceState) {
        ...
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        mThumbnailThread.quit();
        Log.i(TAG, "Background thread destroyed");
    }
    ...
}
```

Assembling a Background Thread 3/3

Hook up ThumbnailDownloader (PhotoGalleryFragment.java)

```
private class GalleryItemAdapter extends ArrayAdapter<GalleryItem> {
@Override
    public View getView(int position, View convertView, ViewGroup parent) {
        ...
        ImageView imageView = (ImageView)convertView
                .findViewById(R.id.gallery_item_imageView);
        imageView.setImageResource(R.drawable.brian_up_close);
        GalleryItem item = getItem(position);
        mThumbnailThread.queueThumbnail(imageView, item.getUrl());
        return convertView;
    }
}
```

Message Anatomy

A message is an instance of **Message** and contains several fields. Three are relevant to your implementation:

what	a user-defined int that describes the message
obj	a user-specified object to be sent with the message
target	the Handler that will handle the message

When you create a **Message**, it will automatically be attached to a **Handler**. And when your **Message** is ready to be processed, **Handler** will be the object in charge of making it happen.

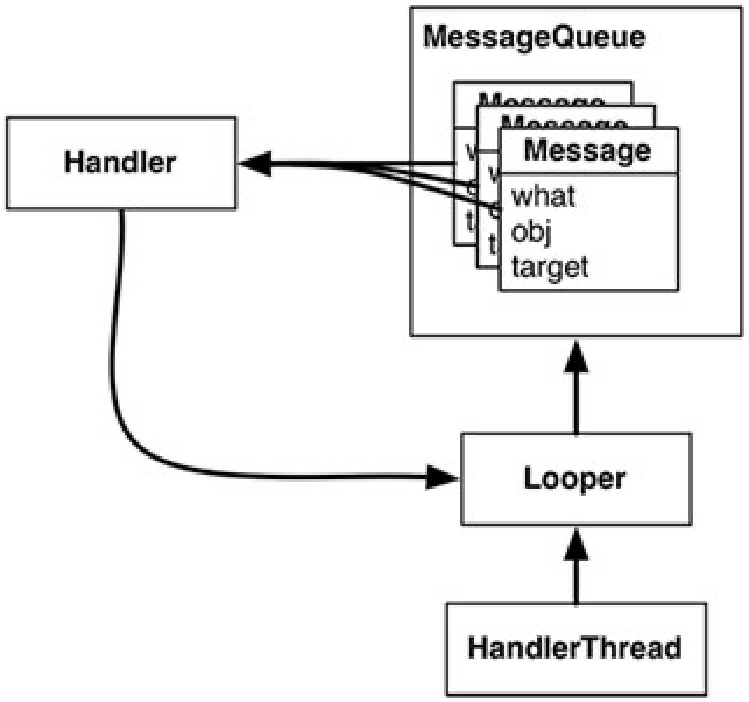
Handler Anatomy 1/2

To do any real work with messages, you will need an instance of the target message Handler first.

A **Handler** is not just a target for processing your Messages. A **Handler** is your interface for creating and posting Messages, too.

Messages must be posted and consumed on a **Looper,** because Looper owns the inbox of Message objects. So Handler always has a reference to its coworker, the Looper.

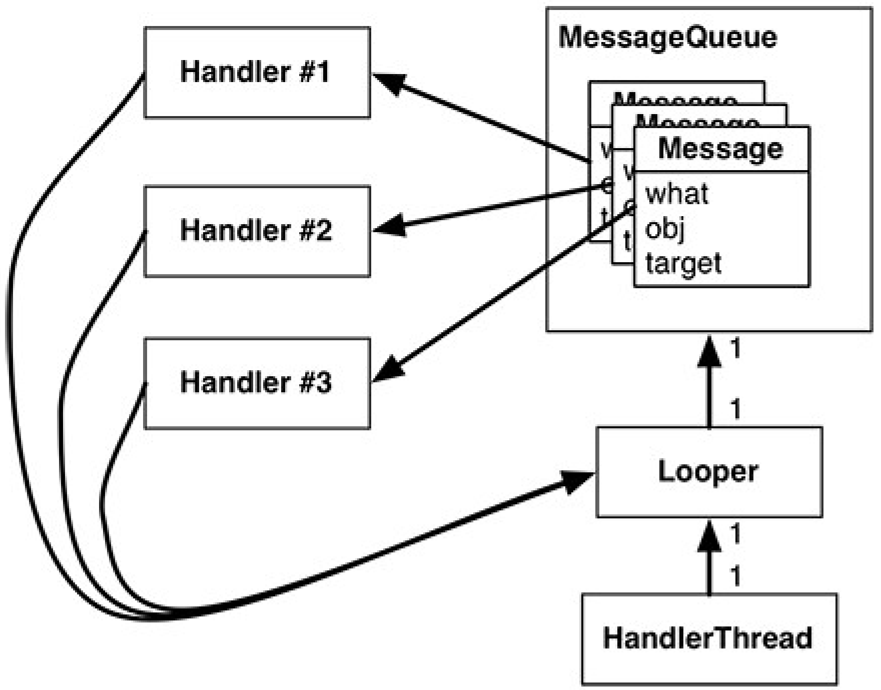
A **Handler** is attached to exactly one Looper, and a Message is attached to exactly one target Handler, called its target. A Looper has a whole queue of Messages.



Looper, Handler, HandlerThread, and Message

Handler Anatomy 2/2

Multiple Handlers can be attached to one Looper. This means that your Handler’s Messages may be living side by side with another Handler’s messages.



Multiple Handlers, one Looper

Using Handlers

Usually, you do not set a message's target **Handler** by hand. It is better to build the message by calling **Handler.obtainMessage(…)**. You pass the other message fields into this method, and it automatically sets the target for you.

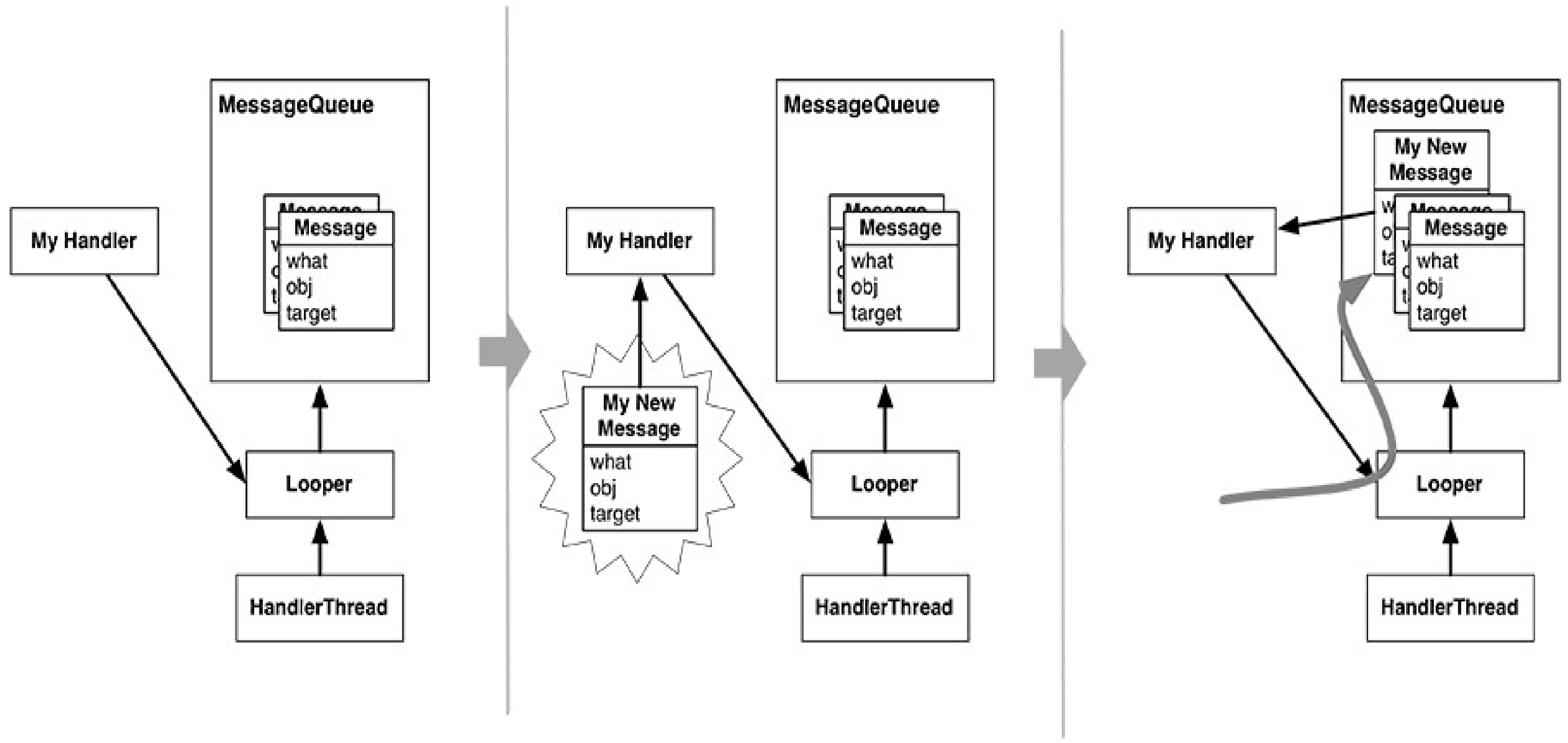
Handler.obtainMessage(…) pulls from a common recycling pool to avoid creating new Message objects, so it is also more efficient than creating new instances.

Once you have obtained a **Message**, you then call **sendToTarget()** to send the **Message** to its **Handler**. The **Handler** will then put the **Message** on the end of **Looper's** message queue.

In this case, you are going to obtain a message and send it to its target within the implementation of **queueThumbnail().** The message's what will be a constant defined as MESSAGE_DOWNLOAD. The obj will be the Token – in this case, the ImageView that the adapter passed in to **queueThumbnail().**

When the looper gets to a particular message in the queue, it gives the message to the message's target to handle. Typically, the message is handled in the target's implementation of **Handler.handleMessage(…).**

Creating a Message and Sending it



In this case, your implementation of **handleMessage(…)** will use **FlickrFetchr** to download bytes from the URL and then turn these bytes into a bitmap.

Obtaining, Sending, and Handling a Message 1/2

```
public class ThumbnailDownloader<Token> extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;
    Handler mHandler;
    Map<Token, String> requestMap =
            Collections.synchronizedMap(new HashMap<Token, String>());
    public ThumbnailDownloader() {    super(TAG);    }
    @SuppressLint("HandlerLeak")
    @Override
    protected void onLooperPrepared() {
        mHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                if (msg.what == MESSAGE_DOWNLOAD) {
                    @SuppressWarnings("unchecked")
                    Token token = (Token)msg.obj;
                    Log.i(TAG, "Got a request for url: " +
                          requestMap.get(token));
                    handleRequest(token);
                }
            }
        };
    }
```

```
public void queueThumbnail(Token token, String url) {
    Log.i(TAG, "Got a URL: " + url");
    requestMap.put(token, url);
    mHandler
        .obtainMessage(MESSAGE_DOWNLOAD, token)
        .sendToTarget();
}
private void handleRequest(final Token token) {
    try {
        final String url = requestMap.get(token);
        if (url == null)  return;
        byte[] bitmapBytes = new FlickrFetchr().getUrlBytes(url);
        final Bitmap bitmap = BitmapFactory
              .decodeByteArray(bitmapBytes, 0, bitmapBytes.length);
        Log.i(TAG, "Bitmap created");
    } catch (IOException ioe) {
        Log.e(TAG, "Error downloading image", ioe);
    }
}
}
```

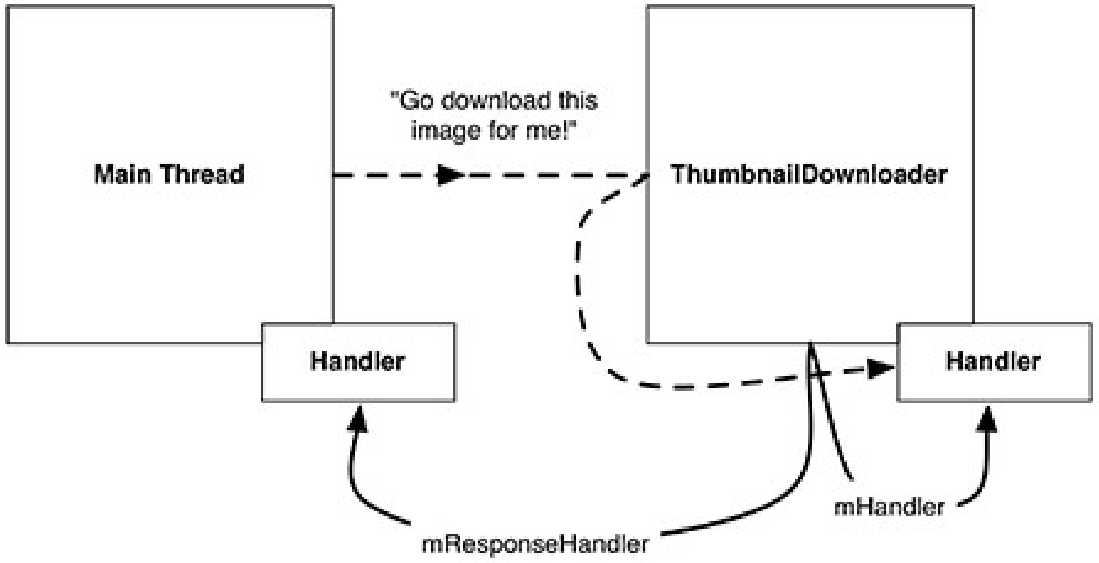
Obtaining, Sending, and Handling a Message 2/2

Handler just printing a log message that *Bitmap is created*. Of course, the request will not be completely handled until you set the bitmap on the **ImageView** that originally came from **GalleryItemAdapter**. However, this is UI work, so it must be done on the main thread.

Everything you have seen so far is using handlers and messages on a single thread – putting messages in your own inbox. Next, you will see how **ThumbnailDownloader** can use a Handler to access the main thread.

Passing Handlers to Update UI 1/2

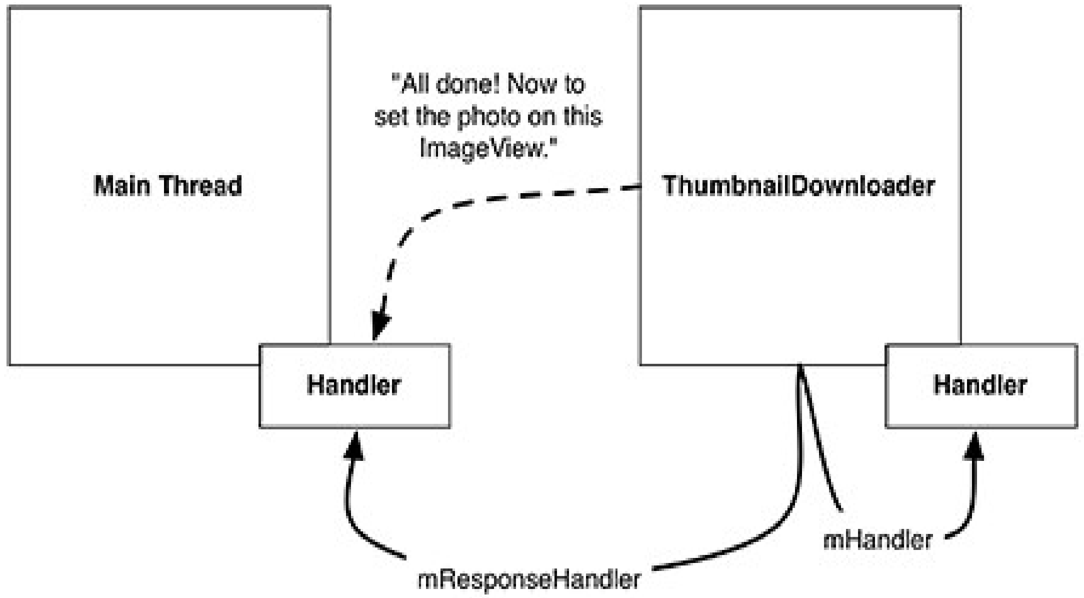
The main thread is a message loop with handlers and a **Looper**. When you create a **Handler** in the main thread, it will be associated with the main thread's **Looper**. You can then pass that **Handler** to another thread. The passed **Handler** maintains its loyalty to the **Looper** of the thread that created it. Any messages the **Handler** is responsible for will be handled on the main thread's queue.



Scheduling work on ThumbnailDownloader from the main thread

Passing Handlers to Update UI 2/2

You can also schedule work on the main thread from the background thread using a **Handler** attached to the main thread.



Scheduling work on the main thread from ThumbnailDownloader's thread

Add Response Handler

(ThumbnailDownloader.java)

```
public class ThumbnailDownloader extends HandlerThread {
    private static final String TAG = "ThumbnailDownloader";
    private static final int MESSAGE_DOWNLOAD = 0;
    Handler mHandler;
    Map<Token,String> requestMap =
            Collections.synchronizedMap(new HashMap<Token,String>());
    Handler mResponseHandler;
    Listener<Token> mListener;
    public interface Listener<Token> {
        void onThumbnailDownloaded(Token token, Bitmap thumbnail);
    }
    public void setListener(Listener<Token> listener) {
        mListener = listener;
    }
    ~~public ThumbnailDownloader() {~~
        ~~super(TAG);~~
    public ThumbnailDownloader(Handler responseHandler) {
        super(TAG);
        mResponseHandler = responseHandler;
    }}
```

Hook up to Response Handler

Hook up to response Handler
(PhotoGalleryFragment.java)

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setRetainInstance(true);
   new FetchItemsTask().execute();
   ~~mThumbnailThread = new ThumbnailDownloader();~~
   mThumbnailThread = new ThumbnailDownloader(new Handler());
   mThumbnailThread.setListener(
       new ThumbnailDownloader.Listener<ImageView>() {
      public void onThumbnailDownloaded(ImageView imageView,
          Bitmap thumbnail) {
         if (isVisible()) {
            imageView.setImageBitmap(thumbnail);
         }
      }
   });
   mThumbnailThread.start();
   mThumbnailThread.getLooper();
   Log.i(TAG, "Background thread started");
}
```

Now **ThumbnailDownloader** has access via mResponseHandler to a **Handler** that is tied to the main thread's Looper. It also has your Listener to do the UI work with the returning Bitmaps.

You could send a custom Message back to the main thread this way. This would require another subclass of **Handler**, with an override of **handleMessage(…).** Instead, let's use another handy Handler method – **post(Runnable).**

Handler.post(Runnable)

Handler.post(Runnable) is a convenience method for posting Messages that look like this:

```
Runnable myRunnable = new Runnable() {
    public void run() {
        /* Your code here */
    }
};
Message m = mHandler.obtainMessage();
m.callback = myRunnable;
```

When a **Message** has its callback field set, instead of being run by its **Handler** target, the **Runnable** in callback is run instead.

In **ThumbnailDownloader.handleRequest()**, add the following code.

Downloading and Displaying Photos

(ThumbnailDownloader.java).

```
..
private void handleRequest(final Token token) {
   try {
      final String url = requestMap.get(token);
      if (url == null)
         return;
      byte[] bitmapBytes = new FlickrFetchr().getUrlBytes(url);
      final Bitmap bitmap = BitmapFactory
            .decodeByteArray(bitmapBytes, 0, bitmapBytes.length);
      Log.i(TAG, "Bitmap created");
      mResponseHandler.post(new Runnable() {
           public void run() {
               if (requestMap.get(token) != url)
                   return;
               requestMap.remove(token);
               mListener.onThumbnailDownloaded(token, bitmap);
           }
       });
   } catch (IOException ioe) {
      Log.e(TAG, "Error downloading image", ioe);
}}
```

Since mResponseHandler is associated with the main thread's Looper, this UI update code will be run on the main thread.

Add Cleanup Method

If the user rotates the screen, **ThumbnailDownloader** may be hanging on to invalid **ImageViews**. Bad things will happen if those **ImageViews** get pressed.

Write the following method to clean all the requests out of your queue.

(ThumbnailDownloader.java)

```
public void clearQueue() {
   mHandler.removeMessages(MESSAGE_DOWNLOAD);
   requestMap.clear();
}
```

Then clean out your downloader in **PhotoGalleryFragment** when your view is destroyed.

Call cleanup method
(PhotoGalleryFragment.java)

```
@Override
public void onDestroyView() {
   super.onDestroyView();
   mThumbnailThread.clearQueue();
}
```

Challenge

A cache is a place to stash a certain number of **Bitmap** objects so that they stick around even when you are done using them. A cache can only hold so many items, so you need a strategy to decide what to keep when your cache runs out of room. Many caches use a strategy called LRU, or “least recently used.” When you are out of room, the cache gets rid of the least recently-used item.

The Android support library has a class called **LruCache** that implements an LRU strategy. For the first challenge, use **LruCache** to add a simple cache to **ThumbnailDownloader**. Whenever you download the **Bitmap** for an URL, you will stick it in the cache. Then, when you are about to download a new image, you will check the cache first to see if you already have it around.

Android Networking
Search

Javed Hasan
BJIT Limited

Search

PhotoGallery Application New Feature

- Searching Flickr

Flickr Search API

- Search API methods, Parameters

The Search Dialog

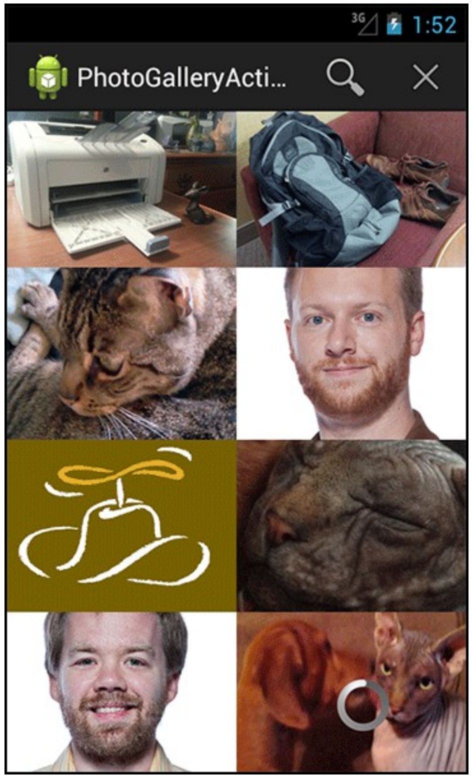
- Creating a Search Interface
- Searchable Activities

How Search Works

- Activity Launch Mode and new Intent
- Saving Search Query with SharedPreference files

PhotoGallery App New Feature

Your next task with **PhotoGallery** is to search photos on Flickr. In this lesson, you will learn how to integrate search into your app the Android way.



Searching Flickr1/3

To search Flickr, you call the flickr.photos.search method.

Here is what a flickr.photos.search method invocation to look for the text "red" looks like:

http://api.flickr.com/services/rest/?method=flickr.photos.search&api_key=XXX&extras=url_s&text=red

This method takes in some new and different parameters to specify what the search terms are, like a text query parameter.

Since both search and getRecent parse **GalleryItems** in the same way, you will refactor some of your old code from fetchItems() into a new method called downloadGalleryItems(String)

Add Flickr search method (FlickrFetchr.java)

```
public class FlickrFetchr {
    public static final String TAG = "PhotoFetcher";
    private static final String ENDPOINT = "http://api.flickr.com/services/rest/";
    private static final String API_KEY = "4f721bbafa75bf6d2cb5af54f937bb70";
    private static final String METHOD_GET_RECENT = "flickr.photos.getRecent";
    private static final String METHOD_SEARCH = "flickr.photos.search";
    private static final String PARAM_EXTRAS = "extras";
    private static final String PARAM_TEXT = "text";
    ...
```

Searching Flickr 2/3

```
~~public ArrayList<GalleryItem> fetchItems() {~~
  public ArrayList<GalleryItem> downloadGalleryItems(String url) {
    ArrayList<GalleryItem> items = new ArrayList<GalleryItem>();
    try {
      ~~String url = Uri.parse(ENDPOINT).buildUpon()~~
          ~~.appendQueryParameter("method", METHOD_GET_RECENT)~~
          ~~.appendQueryParameter("api_key", API_KEY)~~
          ~~.appendQueryParameter(PARAM_EXTRAS, EXTRA_SMALL_URL)~~
          ~~.build().toString();~~
      String xmlString = getUrl(url);
      Log.i(TAG, "Received xml: " + xmlString);
      XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
      XmlPullParser parser = factory.newPullParser();
      parser.setInput(new StringReader(xmlString));
      parseItems(items, parser);
    } catch (IOException ioe) {
      Log.e(TAG, "Failed to fetch items", ioe);
    } catch (XmlPullParserException xppe) {
      Log.e(TAG, "Failed to parse items", xppe);
    }
     return items;
  }
```

Searching Flickr 3/3

```
public ArrayList<GalleryItem> fetchItems() {
        // Move code here from above
        String url = Uri.parse(ENDPOINT).buildUpon()
                .appendQueryParameter("method", METHOD_GET_RECENT)
                .appendQueryParameter("api_key", API_KEY)
                .appendQueryParameter(PARAM_EXTRAS, EXTRA_SMALL_URL)
                .build().toString();
        return downloadGalleryItems(url);
    }
    public ArrayList<GalleryItem> search(String query) {
          String url = Uri.parse(ENDPOINT).buildUpon()
                    .appendQueryParameter("method", METHOD_SEARCH)
                    .appendQueryParameter("api_key", API_KEY)
                    .appendQueryParameter(PARAM_EXTRAS, EXTRA_SMALL_URL)
                    .appendQueryParameter(PARAM_TEXT, query)
                    .build().toString();
          return downloadGalleryItems(url);
     }
}
```

Hardwired Search Query Code

Hook up some test code to call your search code inside PhotoGalleryFragment.FetchItemsTask. For now, you will hardwire a search query just to make sure that it works. Run **PhotoGallery** and see what you get. Hopefully, you will see some Android images.

```
private class FetchItemsTask extends AsyncTask<Void,Void,ArrayList<GalleryItem>> {
    @Override
    protected ArrayList<GalleryItem> doInBackground(Void... params) {
        String query = "android"; // Just for testing

        if (query != null) {
            return new FlickrFetchr().search(query);
        } else {
            return new FlickrFetchr().fetchItems();
        }
    }
    @Override
    protected void onPostExecute(ArrayList<GalleryItem> items) {
        ...
    }
  }
  ...
}
```

Implementing Android’s Search Interface 1/3

You will implement Android’s search interface in **PhotoGallery**. You will start with the old style dialog interface.

Add search strings (res/values/strings.xml)

```
<resources>
   ...
   <string name="title_activity_photo_gallery">
         PhotoGalleryActivity</string>
   <string name="search_hint">Search Flickr</string>
   <string name="search">Search</string>
   <string name="clear_search">Clear Search</string>

</resources>
```

Add search menu items (res/menu/fragment_photo_gallery.xml)

```
<menu
      xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_item_search"
    android:title="@string/search"
    android:icon="@android:drawable/ic_menu_search"
    android:showAsAction="ifRoom"
    />
  <item android:id="@+id/menu_item_clear"
    android:title="@string/clear_search"
    android:icon="@android:drawable/ic_menu_close_clear_cancel"
    android:showAsAction="ifRoom"
    />
</menu>
```

Implementing Android's Search Interface 2/3

Options menu callbacks (PhotoGalleryFragment.java)

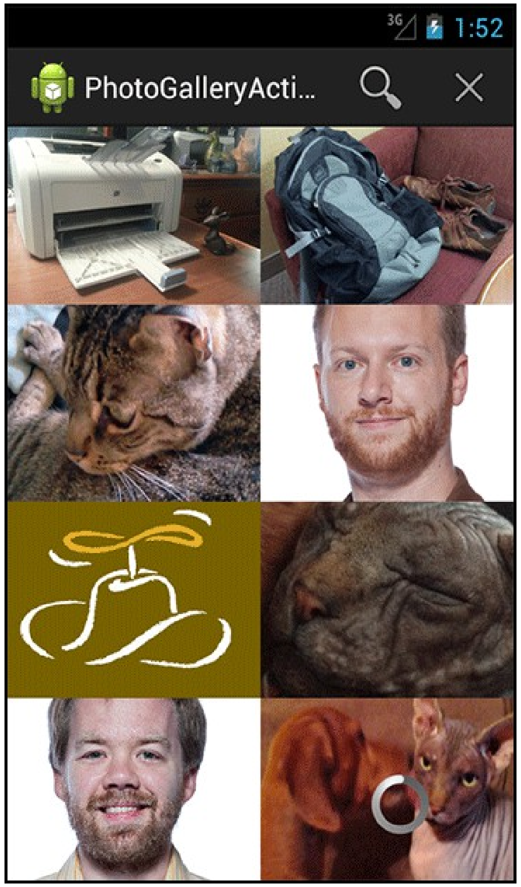
```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
    setHasOptionsMenu(true);
    ...
}
...
@Override
public void onDestroyView() {
    ...
}
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    super.onCreateOptionsMenu(menu, inflater);
    inflater.inflate(R.menu.fragment_photo_gallery, menu);
}
```

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_item_search:
            getActivity().onSearchRequested();
            return true;
        case R.id.menu_item_clear:
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Implementing Android's Search Interface 3/3

Run your new menu interface to see that it displays correctly.

Pressing the search button will not do anything right now, though. For onSearchRequested() to work, you must make **PhotoGalleryActivity** into a *searchable activity.*



Searchable Activities 1/2

There are two things that make an activity searchable. One is an XML file contains an element called searchable that describes how the search dialog should display itself.

The next thing is in **AndroidManifest.xml**. You need to change the *launch mode* of your app, and you need to declare an additional intent filter and a piece of metadata for **PhotoGalleryActivity**.

The intent filter advertises that you can listen to search intents, and the metadata is to attach the XML you just wrote to your activity.

Search configuration (res/xml/searchable.xml)

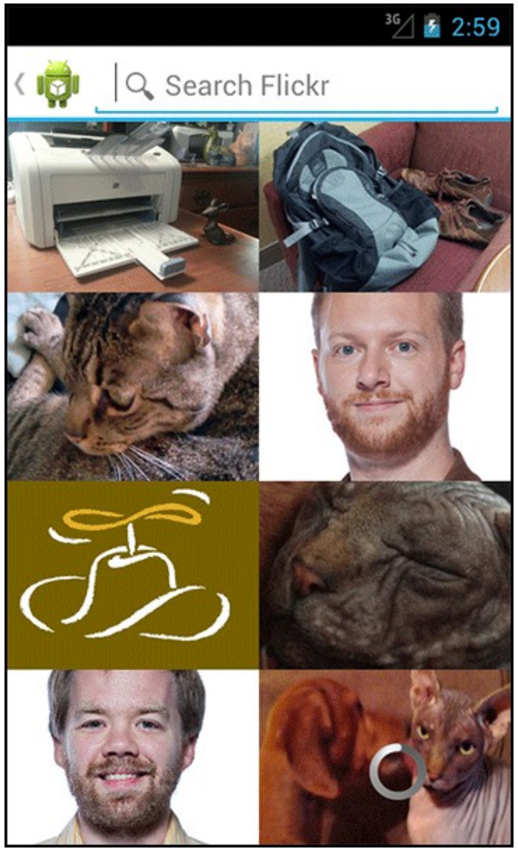
```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
  android:label="@string/app_name"
  android:hint="@string/search_hint"
  />
```

Searchable Activities 2/2

Add intent filter and metadata (AndroidManifest.xml)

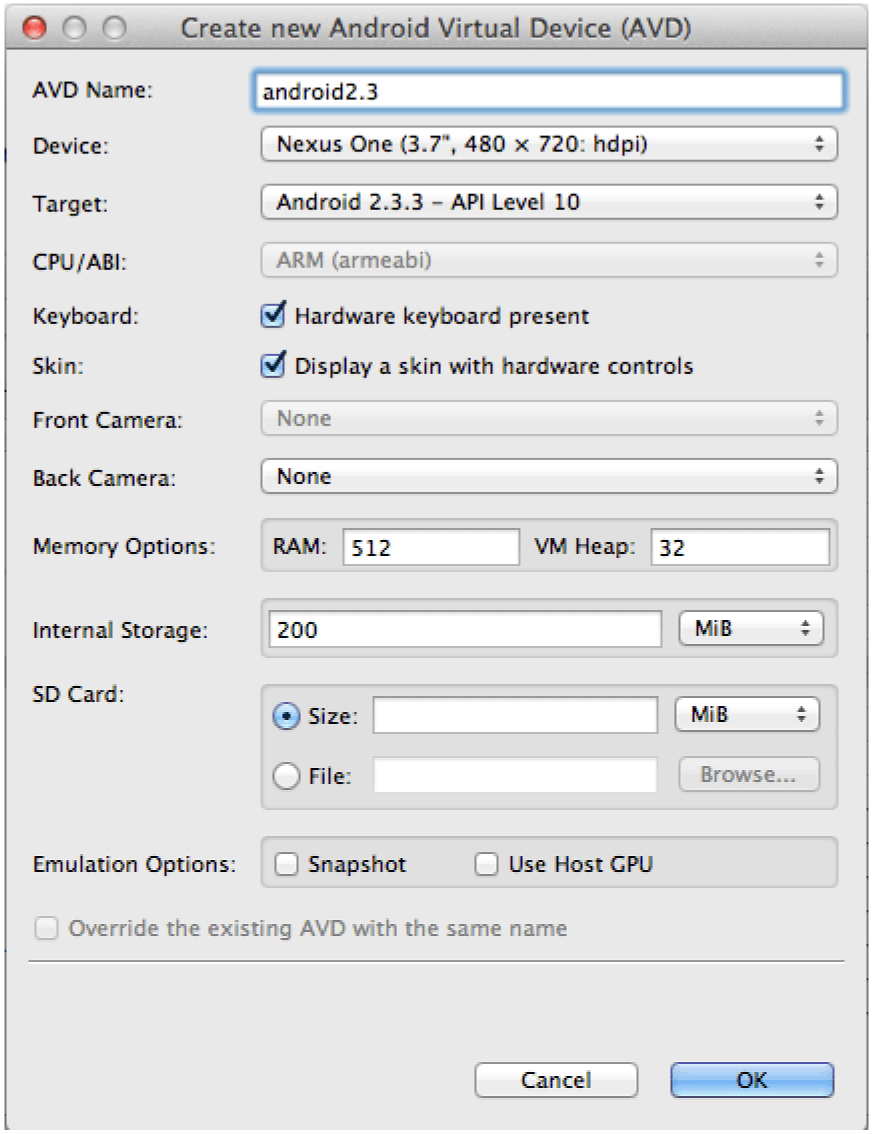
```
<application
  ... >
  <activity
   android:name=".PhotoGalleryActivity"
   android:launchMode="singleTop"
   android:label="@string/title_activity_photo_gallery" >
   <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter>
   <intent-filter>
     <action android:name="android.intent.action.SEARCH" />
    </intent-filter>
    <meta-data android:name="android.app.searchable"
     android:resource="@xml/searchable"/>
  </activity>
 </application>
</manifest>
```

The search dialog



Hardware Search Button

Modify any pre-3.0 emulator to have a hardware search button by configuring your emulator to use the hardware keyboard.



Create new Android Virtual Device (AVD)

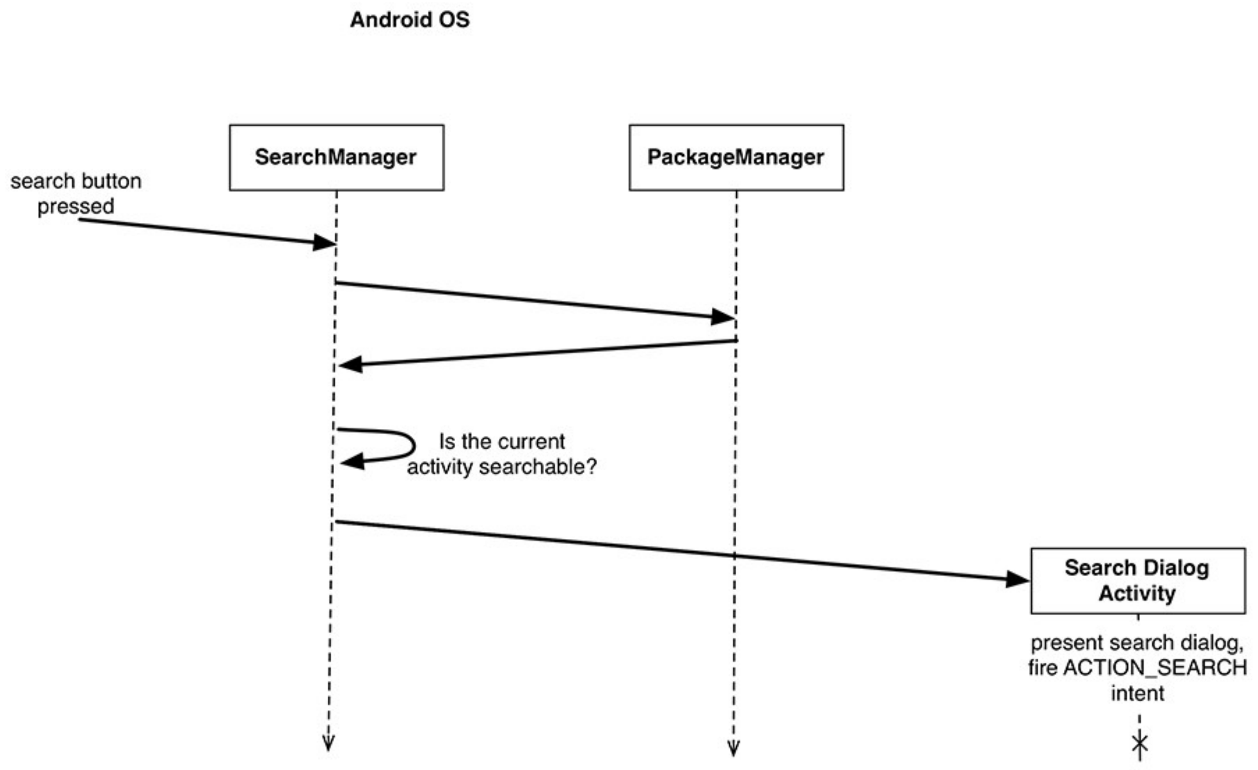
AVD Name:	android2.3
Device:	Nexus One (3.7", 480 × 720: hdpi)
Target:	Android 2.3.3 – API Level 10
CPU/ABI:	ARM (armeabi)
Keyboard:	✓ Hardware keyboard present
Skin:	✓ Display a skin with hardware controls
Front Camera:	None
Back Camera:	None
Memory Options:	RAM: 512 VM Heap: 32
Internal Storage:	200 MiB
SD Card:	● Size: MiB ○ File: Browse...
Emulation Options:	☐ Snapshot ☐ Use Host GPU

☐ Override the existing AVD with the same name

How Search Works 1/3

Normally, pressing the search button will start a new activity. In your case, though, it does not. Why? Because you added android:launchMode="singleTop", which changes your launch mode.

This means that instead of starting a new activity, the search intent you receive will go to your already running **PhotoGalleryActivity** on top of the back stack.


Android OS
SearchManager
PackageManager
search button pressed
Is the current activity searchable?
Search Dialog Activity
present search dialog, fire ACTION_SEARCH intent

How Search Works 2/3

You receive that new intent by overriding onNewIntent(Intent) inside an Activity.

Whenever you receive a new intent, you will want to refresh the items in **PhotoGalleryFragment.**

Refactor **PhotoGalleryFragment** to include an updateItems() method that runs FetchItemsTask to refresh your current items.

Add update method (PhotoGalleryFragment.java)

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);

   setRetainInstance(true);
   ~~new FetchItemsTask().execute();~~
   updateItems();

   mThumbnailThread = new ThumbnailDownloader<ImageView>(new Handler());
   mThumbnailThread.setListener(new ThumbnailDownloader.Listener<ImageView>() {
      ...
   });
   mThumbnailThread.start();
   mThumbnailThread.getLooper();
}

public void updateItems() {
   new FetchItemsTask().execute();
}
```

How Search Works 3/3

Then add your onNewIntent(Intent) override in **PhotoGalleryActivity** to receive the new intent and refresh your **PhotoGalleryFragment's** items:

Override onNewIntent(…) (PhotoGalleryActivity.java)

```
public class PhotoGalleryActivity extends SingleFragmentActivity {
   private static final String TAG = "PhotoGalleryActivity";
   @Override
   public Fragment createFragment() {
      return new PhotoGalleryFragment();
   }
   @Override
   public void onNewIntent(Intent intent) {
      PhotoGalleryFragment fragment = (PhotoGalleryFragment)
         getSupportFragmentManager().findFragmentById(R.id.fragmentContainer);
      if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
         String query = intent.getStringExtra(SearchManager.QUERY);
         Log.i(TAG, "Received a new search query: " + query);
      }
      fragment.updateItems();
   }
}
```

You should be able to see **PhotoGalleryActivity** receiving the new intent now in *LogCat* when you run a search.

Integrate that Search Query into Your App

Next up is to integrate that search query into your app. You will be implementing search so that there is only a single search query at any given time.

It would be nice if that query were persistent. For simple values persistence, though, shared preferences are simpler to implement and better behaved.

Shared preferences are files on your file system that you read and edit using the **SharedPreferences** class. An instance of **SharedPreferences** acts like a key-value store, much like **Bundle**, except that it is backed by persistent storage. The keys are strings, and the values are atomic data types.

Shared preferences constant (FlickrFetchr.java)

```
public class FlickrFetchr {
   public static final String TAG = "FlickrFetchr";

   public static final String PREF_SEARCH_QUERY = "searchQuery";

   private static final String ENDPOINT = "http://api.flickr.com/services/rest/";
   ...
```

Save Query String into Shared Preference

Use the `PreferenceManager.getDefaultSharedPreferences(Context)` method, which returns an instance of **SharedPreference** with a default name and private permissions.

Save out search query (PhotoGalleryActivity.java)

```
@Override
public void onNewIntent(Intent intent) {
    PhotoGalleryFragment fragment = (PhotoGalleryFragment)getSupportFragmentManager()
            .findFragmentById(R.id.fragmentContainer);

    if (Intent.ACTION_SEARCH.equals(intent.getAction())) {
        String query = intent.getStringExtra(SearchManager.QUERY);
        Log.i(TAG, "Received a new search query: " + query);

        PreferenceManager.getDefaultSharedPreferences(this)
                .edit()
                .putString(FlickrFetchr.PREF_SEARCH_QUERY, query)
                .commit();
    }
    fragment.updateItems();
}
```

Read Query String from Shared Preference

Getting a value you previously stored is as simple as calling SharedPreferences.getString(…), getInt(…), or whichever method is appropriate for your data type.

Add code to **PhotoGalleryFragment** to fetch your search query from the default **SharedPreferences.**

```
private class FetchItemsTask extends AsyncTask<Void,Void,ArrayList<GalleryItem>> {
   @Override
   protected ArrayList<GalleryItem>  doInBackground(Void...  params) {
     ~~String query = "android"; // just for testing~~
     Activity activity = getActivity();
     if (activity == null)
        return new ArrayList<GalleryItem>();
     String query = PreferenceManager.getDefaultSharedPreferences(activity)
        .getString(FlickrFetchr.PREF_SEARCH_QUERY,  null);
     if (query != null) {
        return new FlickrFetchr().search(query);
     } else {
        return new FlickrFetchr().fetchItems();
     }
   }
   @Override
   protected void onPostExecute(ArrayList<GalleryItem>  items) {
     ...
   }
}
```

Implement Cancel

To implement canceling a search, clear out your search term from your shared preferences and call updateItems() again:

(PhotoGalleryFragment.java)

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        ...
        case R.id.menu_item_clear:
            PreferenceManager.getDefaultSharedPreferences(getActivity())
                    .edit()
                    .putString(FlickrFetchr.PREF_SEARCH_QUERY, null)
                    .commit();
               updateItems();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Using SearchView on Post-Android 3.0

Honeycomb added a new class called **SearchView. SearchView** is an action view – a view that may be included within the action bar.

SearchView allows your entire search interface to take place within your activity's action bar, instead of inside a dialog superimposed on your activity.

Add an action view to your menu (res/menu/fragment_photo_gallery.xml)

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_item_search"
    android:title="@string/search"
    android:icon="@android:drawable/ic_menu_search"
    android:showAsAction="ifRoom"
    android:actionViewClass="android.widget.SearchView"
    />
  <item android:id="@+id/menu_item_clear"
    ...
    />
</menu>
```

SearchView does not generate any onOptionsItemSelected(…) callbacks. This is a good thing because it means you can leave those callbacks in place for older devices that do not support action views.

Add a SearchView to Your Menu

Honeycomb added a new class called **SearchView. SearchView** is an action view – a view that may be included within the action bar.

SearchView allows your entire search interface to take place within your activity's action bar, instead of inside a dialog superimposed on your activity.

Add an action view to your menu (res/menu/fragment_photo_gallery.xml)

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_item_search"
    android:title="@string/search"
    android:icon="@android:drawable/ic_menu_search"
    android:showAsAction="ifRoom"
    android:actionViewClass="android.widget.SearchView"
    />
  <item android:id="@+id/menu_item_clear"
    ...
    />
</menu>
```

SearchView does not generate any onOptionsItemSelected(…) callbacks. This is a good thing because it means you can leave those callbacks in place for older devices that do not support action views.

Set Search Config Info to SearchView

SearchView needs to know what your search configuration is before it will send you any search intents. You need to add some code to onCreateOptionsMenu(…) that pulls out your search configuration and sends it to **SearchView**.

Configure SearchView (PhotoGalleryFragment.java)

```
@Override
@TargetApi(11)
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
   super.onCreateOptionsMenu(menu, inflater);
   inflater.inflate(R.menu.fragment_photo_gallery, menu);
   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
      // Pull out the SearchView
      MenuItem searchItem = menu.findItem(R.id.menu_item_search);
      SearchView searchView = (SearchView)searchItem.getActionView();
       // Get the data from our searchable.xml as a SearchableInfo
       SearchManager searchManager = (SearchManager)getActivity()
           .getSystemService(Context.SEARCH_SERVICE);
       ComponentName name = getActivity().getComponentName();
       SearchableInfo searchInfo = searchManager.getSearchableInfo(name);
       searchView.setSearchableInfo(searchInfo);
   }
}
```

The Search within Your Activity

SearchManager is a system service that is responsible for all things search related.

All the information about a search, including the name of the activity that should get the intent and everything in your *searchable.xml,* is stashed in the **SearchableInfo** object you get by calling `getSearchableInfo(ComponentName)` here.

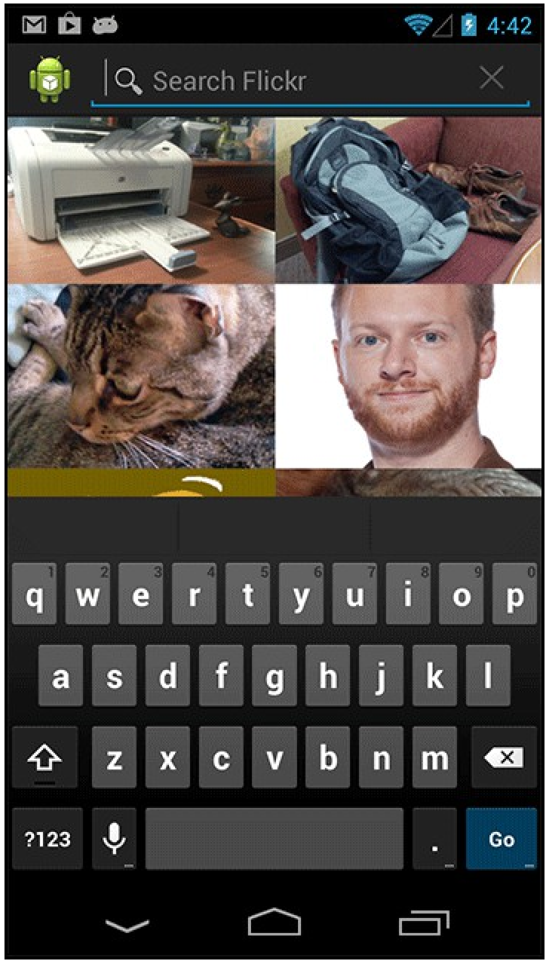
Once you have your **SearchableInfo,** you tell **SearchView** about it by calling `setSearchableInfo(SearchableInfo)`**.**

And now your SearchView is totally wired.

Run and search on a post-3.0 device to see it work.

Once SearchView is properly configured, it behaves exactly like your earlier searches did.

Except for one minor detail: if you try this using the hardware keyboard on an emulator, you will see the search executed two times, one after the other.



Android Networking
Background Services

Javed Hasan

BJIT Limited

Background Services

PhotoGallery Application New Feature

- Looking for new search results in background

Creating an Intent Service

- How Intent Service commands
- What Services are for: Safe background networking

Delayed Execution with Alarm Manager

- Managing Alarms with Pending Intent
- Notifications

PhotoGallery App New Feature

You will add a new feature to PhotoGallery that will allow users to poll for new search results in the background.

Whenever a new search result is available, the user will receive a notification in the status bar.

If you need to do something out of sight and out of mind, like play music or check for new blog posts on an RSS feed, then you need a *Service*.

Creating an IntentService

IntentService. What does it do? Well, it is sort of like an activity. It is a context (Service is a subclass of Context), and it responds to intents (as you can see in **onHandleIntent(Intent)).**

Create a subclass of **IntentService** called **PollService**. This will be the service you use to poll for search results.

PollService's onHandleIntent(Intent) method will be automatically stubbed out for you.

Fill **onHandleIntent(Intent)** out with a log statement, add a log tag, and define a default constructor.

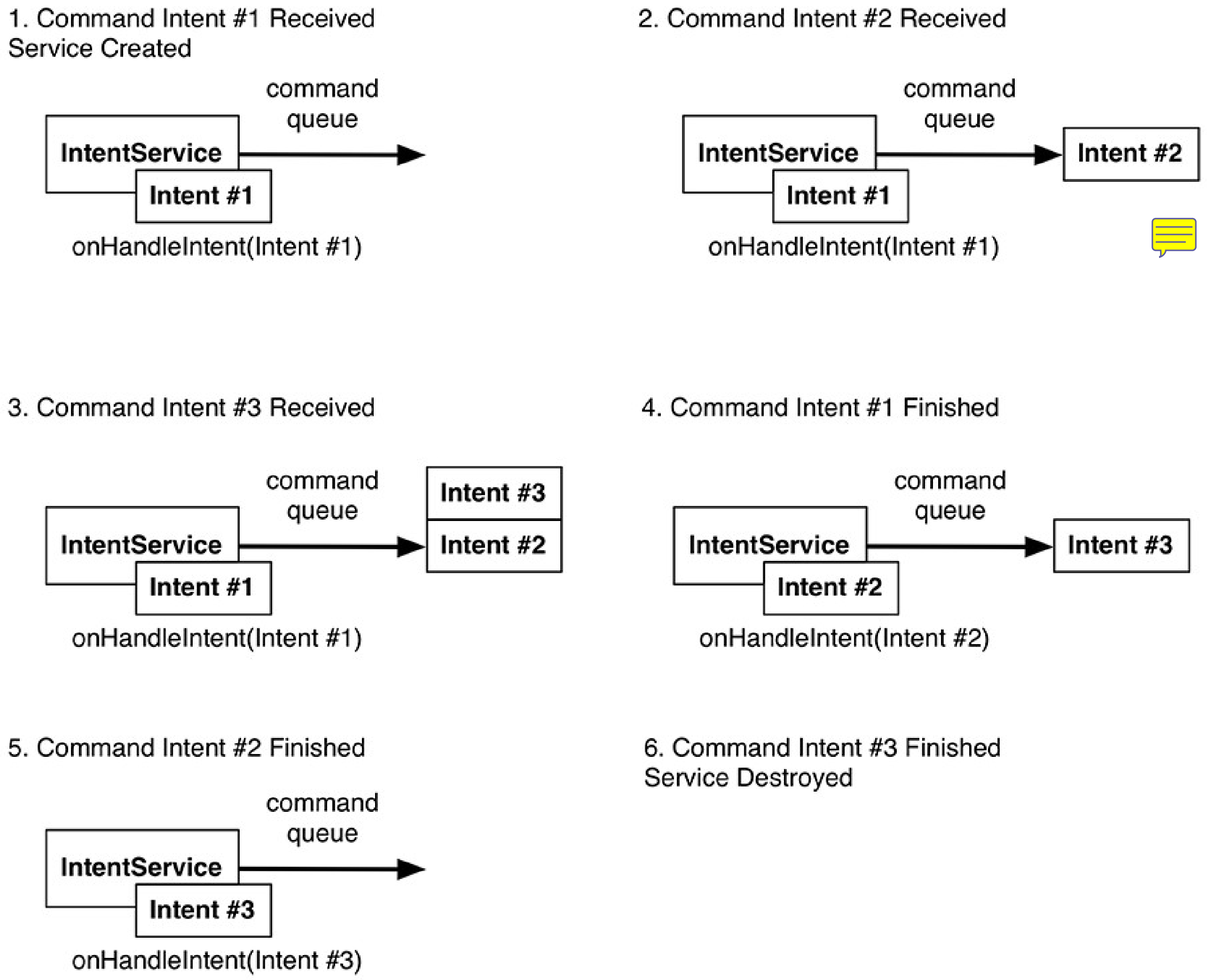
Create PollService (PollService.java)

```
public class PollService extends IntentService {
    private static final String TAG = "PollService";
    public PollService() {
        super(TAG);
    }
    @Override
    protected void onHandleIntent(Intent intent) {
        Log.i(TAG, "Received an intent: " + intent);
    }
}
```

A service's intents are called *commands*. Each command is an instruction to the service to do so.

Depending on the kind of service, that command could be serviced in a variety of ways.

How IntentService Services Commands


1. Command Intent #1 Received
Service Created
command queue
IntentService
Intent #1
onHandleIntent(Intent #1)
2. Command Intent #2 Received
command queue
IntentService
Intent #1
Intent #2
onHandleIntent(Intent #1)
3. Command Intent #3 Received
command queue
IntentService
Intent #1
Intent #3
Intent #2
onHandleIntent(Intent #1)
4. Command Intent #1 Finished
command queue
IntentService
Intent #2
Intent #3
onHandleIntent(Intent #2)
5. Command Intent #2 Finished
command queue
IntentService
Intent #3
onHandleIntent(Intent #3)
6. Command Intent #3 Finished
Service Destroyed

Creating an IntentService

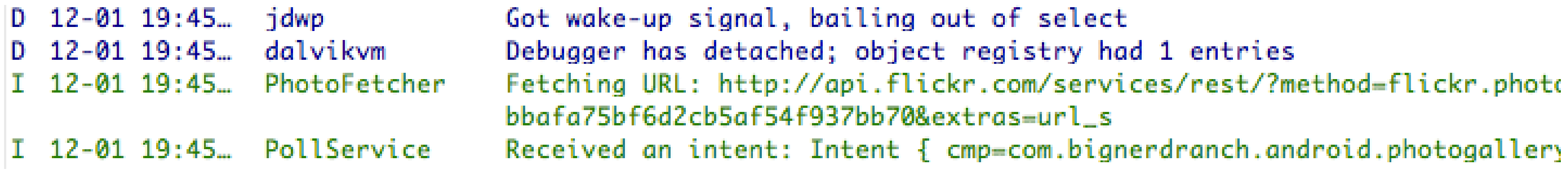
Since services, like activities, respond to intents, they must also be declared in your AndroidManifest.xml.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 ... >
 ...
 <application .. >
  <activity
   android:name=".PhotoGalleryActivity" ... >
   ...
  </activity>
  <service android:name=".PollService" />
 </application> </manifest>
```

Add service startup code (PhotoGalleryFragment.java)

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setRetainInstance(true);
  setHasOptionsMenu(true);
  updateItems();
  Intent i = new Intent(getActivity(), PollService.class);
  getActivity().startService(i);
  mThumbnailThread =
    new ThumbnailDownloader<ImageView>(new Handler());
  ...
}
```

Your service’s first steps



```
D  12-01 19:45…  jdwp          Got wake-up signal, bailing out of select
D  12-01 19:45…  dalvikvm      Debugger has detached; object registry had 1 entries
I  12-01 19:45…  PhotoFetcher  Fetching URL: http://api.flickr.com/services/rest/?method=flickr.phot
                               bbafa75bf6d2cb5af54f937bb70&extras=url_s
I  12-01 19:45…  PollService   Received an intent: Intent { cmp=com.bignerdranch.android.photogaller
```

Safe Background Networking

To perform networking in the background safely, you need to verify with the **ConnectivityManager** that the network is available.

Check for background network availability (PollService.java)

```
@Override
public void onHandleIntent(Intent intent) {
   ConnectivityManager cm = (ConnectivityManager)
         getSystemService(Context.CONNECTIVITY_SERVICE);
   @SuppressWarnings("deprecation")
   boolean isNetworkAvailable =
         cm.getBackgroundDataSetting() &&
         cm.getActiveNetworkInfo() != null;
    if (!isNetworkAvailable) return;
   Log.i(TAG, "Received an intent: " + intent);
}
```

Acquire network state permission (AndroidManifest.xml)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.bignerdranch.android.photogallery"
  android:versionCode="1"
  android:versionName="1.0" >
  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="17" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission
         android:name="android.permission.ACCESS_NETWORK_STATE" />
  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    ...
  </application>
</manifest>
```

Looking for New Search Results 1/2

Your service will be polling for new results, so it will need to know what the last result fetched was. Here is what you need to do:

1. Pull out the current query and the last result ID from the default **SharedPreferences**.
2. Fetch the latest result set with **FlickrFetchr**.
3. If there are results, grab the first one.
4. Check to see if it is different from the last result ID.
5. Store the first result back in **SharedPreferences**.

Add recent ID preference constant (FlickrFetchr.java)

```
public class FlickrFetchr {
    public static final String TAG = "PhotoFetcher";

    public static final String PREF_SEARCH_QUERY = "searchQuery";
    public static final String PREF_LAST_RESULT_ID = "lastResultId";

    private static final String ENDPOINT = "http://api.flickr.com/services/rest/";
    private static final String API_KEY = "xxx";
```

Looking for New Search Results 2/2

Checking for new results (PollService.java)

```
@Override
protected void onHandleIntent(Intent intent) {
    ...
    if (!isNetworkAvailable) return;
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(this);
    String query = prefs.getString(FlickrFetchr.PREF_SEARCH_QUERY, null);
    String lastResultId = prefs.getString(FlickrFetchr.PREF_LAST_RESULT_ID, null);
    ArrayList<GalleryItem> items;
    if (query != null) {   items = new FlickrFetchr().search(query); }
    else { items = new FlickrFetchr().fetchItems(); }
     if (items.size() == 0)   return;
    String resultId = items.get(0).getId();
    if (!resultId.equals(lastResultId)) {
        Log.i(TAG, "Got a new result: " + resultId);
     } else {
        Log.i(TAG, "Got an old result: " + resultId);
     }
    prefs.edit()
        .putString(FlickrFetchr.PREF_LAST_RESULT_ID, resultId)
        .commit();
}
```

Delayed Execution with AlarmManager 1/2

To actually use your service in the background, you will need some way to make things happen when none of your activities are running. Say, by making a timer that goes off every five minutes or so.

You will use **AlarmManager** for this purpose.

AlarmManager is a system service that can send Intents for you.

How do you tell **AlarmManager** what intents to send?

You use a **PendingIntent**. You can use **PendingIntent** to package up a wish: “I want to start PollService.” You can then send that wish to other components on the system, like **AlarmManager**.

```
public class PollService extends IntentService {
   private static final String TAG = "PollService";
   private static final int POLL_INTERVAL = 1000 * 15; // 15 seconds
   public PollService() {
      super(TAG);
   }
   @Override
   public void onHandleIntent(Intent intent) { ...}
  public static void setServiceAlarm(Context context, boolean isOn) {
       Intent i = new Intent(context, PollService.class);
       PendingIntent pi = PendingIntent.getService( context, 0, i, 0);
       AlarmManager alarmManager = (AlarmManager)
               context.getSystemService(Context.ALARM_SERVICE);
       if (isOn) {
          alarmManager.setRepeating(AlarmManager.RTC,
                 System.currentTimeMillis(), POLL_INTERVAL, pi);
       } else {
          alarmManager.cancel(pi);
          pi.cancel();
       }
 }}
```

Delayed Execution with AlarmManager 2/2

Add alarm startup code (PhotoGalleryFragment.java)

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setRetainInstance(true);
   setHasOptionsMenu(true);
   updateItems();
  ~~Intent i = new Intent(getActivity(), PollService.class);~~
  ~~getActivity().startService(i);~~
   PollService.setServiceAlarm(getActivity(), true);
   mThumbnailThread =
        new ThumbnailDownloader<ImageView>(new Handler());
   ...
}
```

Run PhotoGallery. Then immediately hit the back button and exit out of the app.

Notice anything in LogCat?

PollService is faithfully chugging along, running again every 15 seconds.

This is what **AlarmManager** is designed to do.

Even if your process gets shut down, **AlarmManager** will keep on firing intents to start **PollService** again and again.

Pending Intent

A **PendingIntent** is a token object. When you get one here by calling **PendingIntent.getService(…)**, you say to the OS, “Please remember that I want to send this intent with **startService(Intent)**.”

Later on you can call **send()** on your **PendingIntent** token, and the OS will send the intent you originally wrapped up in exactly the way you asked.

The really nice thing about this is that when you give that **PendingIntent** token to someone else and they use it, it sends that token as your application.

Also, since the **PendingIntent** itself lives in the OS, not in the token, you maintain control of it. If you wanted to be cruel, you could give someone else a **PendingIntent** object and then immediately cancel it, so that **send()** does nothing.

If you request a **PendingIntent** twice with the same intent, you will get the same **PendingIntent**. You can use this to test whether a **PendingIntent** already exists or to cancel a previously issued **PendingIntent**.

Managing Pending Intent

You can only register one alarm for each **PendingIntent.**

Since the **PendingIntent** is also cleaned up when the alarm is canceled, you can check whether that **PendingIntent** exists or not to see whether the alarm is active or not.

This is done by passing in the **PendingIntent.FLAG_NO_CREATE** flag to **PendingIntent.getService(…).**

This flag says that if the **PendingIntent** does not already exist, return null instead of creating it.

Add isServiceAlarmOn() method (PollService.java)

```
public static void setServiceAlarm(Context context, boolean isOn)
{
   ...
}
public static boolean isServiceAlarmOn(Context context) {
    Intent i = new Intent(context, PollService.class);
    PendingIntent pi = PendingIntent.getService(
            context, 0, i, PendingIntent.FLAG_NO_CREATE);
    return pi != null;
}
```

Controlling Your Alarms

Let’s add a menu interface to turn Alarm on and off.
Add service toggle
(menu/fragment_photo_gallery.xml)

```
<menu
      xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/menu_item_search"
    android:title="@string/search"
    android:icon="@android:drawable/ic_menu_search"
    android:showAsAction="ifRoom"
    android:actionViewClass="android.widget.SearchView"
    />
  <item android:id="@+id/menu_item_clear"
    android:title="@string/clear_search"
    android:icon="@android:drawable/ic_menu_close_clear_cancel"
    android:showAsAction="ifRoom"
    />
  <item android:id="@+id/menu_item_toggle_polling"
    android:title="@string/start_polling"
    android:showAsAction="ifRoom"
    />
</menu>
```

Add polling strings
(res/values/strings.xml)

```
<resources>
   ...
   <string name="search">Search</string>
   <string name="clear_search">Clear Search</string>
   <string name="start_polling">Poll for new pictures</string>
   <string name="stop_polling">Stop polling</string>
   <string name="new_pictures_title">
         New PhotoGallery Pictures</string>
   <string name="new_pictures_text">
        You have new pictures in PhotoGallery.</string>
</resources>
```

Toggle Menu Item Implementation

Delete your old debug code for starting your alarm, and add in implementation for your menu item.

```
@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setRetainInstance(true);
   setHasOptionsMenu(true);
   updateItems();
   ~~PollService.setServiceAlarm(getActivity(), true);~~
   mThumbnailThread = new ThumbnailDownloader<ImageView>
(new Handler());
   ...
}
...
```

```
@Override
@TargetApi(11)
public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      case R.id.menu_item_search:
         ...
      case R.id.menu_item_clear:
         ...
         updateItems();
         return true;
      case R.id.menu_item_toggle_polling:
         boolean shouldStartAlarm = !PollService.
isServiceAlarmOn(getActivity());
         PollService.setServiceAlarm(getActivity(),
shouldStartAlarm);
         return true;
      default:
         return super.onOptionsItemSelected(item);
   }
}
```

Updating Options Menu Items 1/2

If you need to update the contents of an options menu item, you should put that code in **onPrepareOptionsMenu(Menu).**

This method is called every single time the menu needs to be configured, not just when it is created the first time.

Add onPrepareOptionsMenu(Menu) (PhotoGalleryFragment.java)

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    ...
}
@Override
public void onPrepareOptionsMenu(Menu menu) {
    super.onPrepareOptionsMenu(menu);
    MenuItem toggleItem = menu.findItem(R.id.menu_item_toggle_polling);
    if (PollService.isServiceAlarmOn(getActivity())) {
        toggleItem.setTitle(R.string.stop_polling);
    } else {
        toggleItem.setTitle(R.string.start_polling);
    }
}
```

Updating Options Menu Items 2/2

In pre-3.0 devices, this method is called every time the menu is displayed, which ensures that your menu item always shows the right text.

After 3.0, the action bar does not automatically update itself. You have to manually tell it to call **onPrepareOptionsMenu(Menu)** and refresh its items by calling **Activity.invalidateOptionsMenu().**

Invalidate your options menu (PhotoGalleryFragment.java)

```
@Override
@TargetApi(11)
public boolean onOptionsItemSelected(MenuItem item) {
   switch (item.getItemId()) {
      ...
      case R.id.menu_item_toggle_polling:
         boolean shouldStartAlarm = !PollService.isServiceAlarmOn(getActivity());
         PollService.setServiceAlarm(getActivity(), shouldStartAlarm);
         if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB)
               getActivity().invalidateOptionsMenu();
         return true;
      default:
         return super.onOptionsItemSelected(item);
   }
}
```

Notifications

When your service needs to communicate something to the user, the proper tool is almost always a *notification*.

Notifications are items that appear in the notifications drawer, which the user can access by dragging it down from the top of the screen.

To post a notification, you first need to create a **Notification** object. **Notifications** are created by using a builder object, much like **AlertDialog**.

At a minimum, your **Notification** should have:

- *ticker text* to display in the status bar when the notification is first shown
- an *icon* to show in the status bar after the ticker text goes away
- a *view* to show in the notification drawer to represent the notification itself
- a *PendingIntent* to fire when the user presses the notification in the drawer

Add a Notification 1/2

(PollService.java)

```
@Override
public void onHandleIntent(Intent intent) {
    ...
    String resultId = items.get(0).getId();
    if (!resultId.equals(lastResultId)) {
        Log.i(TAG, "Got a new result: " + resultId);
        Resources r = getResources();
        PendingIntent pi = PendingIntent
            .getActivity(this, 0, new Intent(this, PhotoGalleryActivity.class), 0);
        Notification notification = new NotificationCompat.Builder(this)
            .setTicker(r.getString(R.string.new_pictures_title))
            .setSmallIcon(android.R.drawable.ic_menu_report_image)
            .setContentTitle(r.getString(R.string.new_pictures_title))
            .setContentText(r.getString(R.string.new_pictures_text))
            .setContentIntent(pi)
            .setAutoCancel(true)
            .build();
```

Add a Notification 2/2

```
    NotificationManager notificationManager = (NotificationManager)
            getSystemService(NOTIFICATION_SERVICE);
        notificationManager.notify(0, notification);
    }
    prefs.edit()
        .putString(FlickrFetchr.PREF_LAST_RESULT_ID, resultId)
        .commit();
}
```

Change to a sensible alarm constant (PollService.java)

```
public class PollService extends IntentService {
    private static final String TAG = "PollService";
    ~~public static final int POLL_INTERVAL = 1000 * 15; // 15 seconds~~
    public static final int POLL_INTERVAL = 1000 * 60 * 5; // 5 minutes
    public PollService() {
        super(TAG);
    }
```

Android Networking
Broadcast Intents

Javed Hasan
BJIT Limited

Broadcast Intents

PhotoGallery Application New Feature

- Keep background service running even system is booted
- Show background notification only

Waking up on Boot

- Declare Broadcast Receivers in the manifest
- How to use receivers

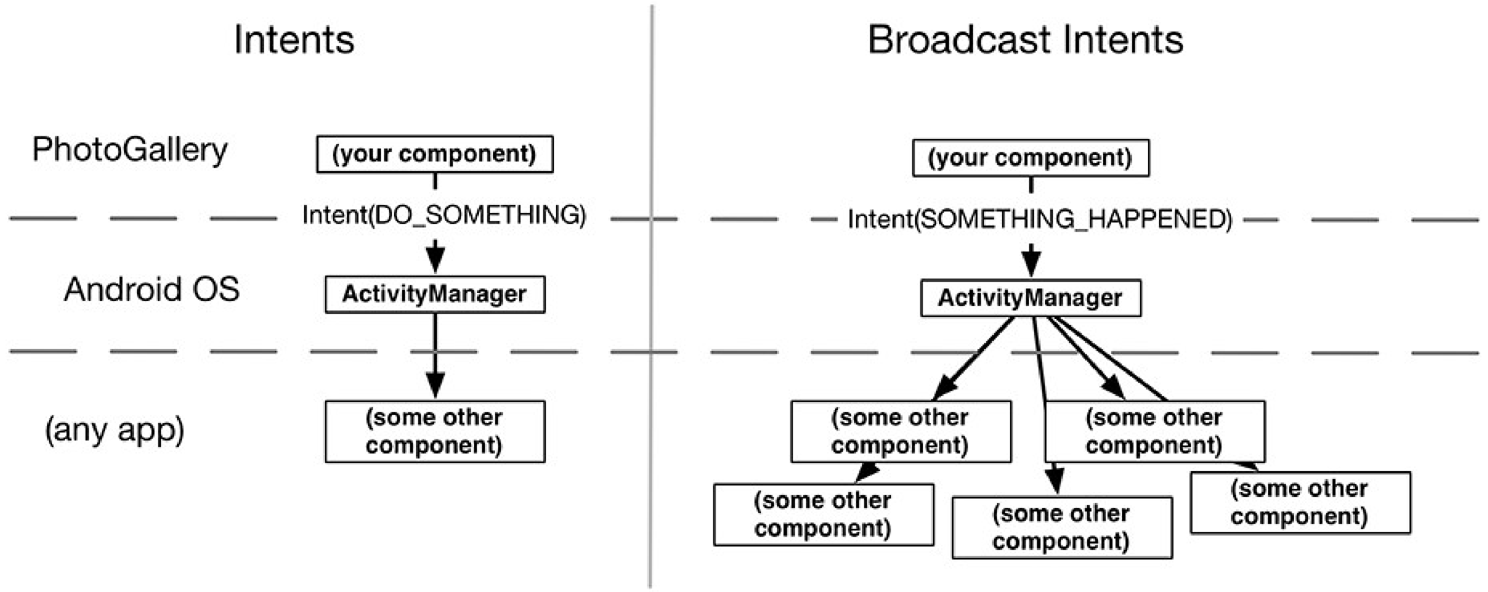
Filtering Foreground Notifications

- Sending broadcast intents
- Dynamic broadcast receivers
- Using private permissions
- Receiving results with ordered broadcasts

Broadcast Intents Overview

Things are happening all the time on an Android device. WiFi is going in and out of range, packages are getting installed, phone calls and text messages are coming in.

When many components on the system need to know that some event has occurred, Android uses a broadcast intent to tell everyone about it. Broadcast intents are received by *broadcast receivers*.



Regular Intents vs. Broadcast Intents

Waking Up on a Boot 1/4

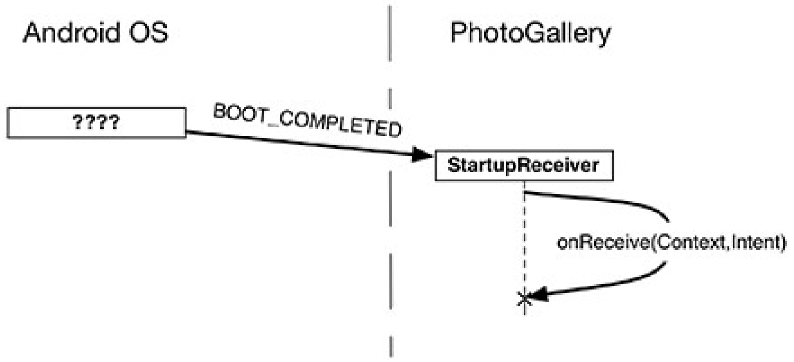
PhotoGallery’s background alarm works, but it is not perfect. If the user reboots their phone, the alarm will be forgotten.

Apps that perform an ongoing process for the user usually need to wake themselves up after the device is booted. You can detect when boot is completed by listening for a broadcast intent with the BOOT_COMPLETED action.

Your first broadcast receiver (StartupReceiver.java)

```
package com.bignerdranch.android.photogallery;
...
public class StartupReceiver extends BroadcastReceiver {
    private static final String TAG = "StartupReceiver";
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i(TAG, "Received broadcast intent: " + intent.getAction());
    }
}
```

Receiving BOOT_COMPLETED



Waking Up on a Boot 2/4

Adding your receiver to the manifest (AndroidManifest.xml)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.bignerdranch.android.photogallery"
 ... >
 ...
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
  <application
    ... >
    <activity
      ... >
      ...
    </activity>
    <service android:name=".PollService" />
    <receiver android:name=".StartupReceiver">
         <intent-filter>
              <action android:name="android.intent.action.BOOT_COMPLETED" />
         </intent-filter>
     </receiver>
  </application>
</manifest>
```

With your broadcast receiver declared in your manifest, it will wake up anytime a matching broadcast intent is sent – even if your app is not currently running.

Upon waking up, the ephemeral broadcast receiver's **onReceive(Context,Intent)** method will be run, and then it will die.

Waking Up on a Boot 3/4

onReceive(Context,Intent) runs on your main thread, so you cannot do any heavy lifting inside it. That means no networking or heavy work with permanent storage.

Your recurring alarm needs to be reset when the system finishes starting. Receivers are suitable to do such light works.

Your receiver will need to know whether the alarm should be on or off.

Add a preference constant to **PollService** to store this information. Then your **StartupReceiver** can use it to turn the alarm on at boot.

```
public class PollService extends IntentService {
   private static final String TAG = "PollService";
   private static final int POLL_INTERVAL = 1000 * 60 * 5; // 5 minutes
   public static final String PREF_IS_ALARM_ON = "isAlarmOn";
   ...
   public static void setServiceAlarm(Context context, boolean isOn) {
      Intent i = new Intent(context, PollService.class);
      PendingIntent pi = PendingIntent.getService( context, 0, i, 0);
      AlarmManager alarmManager = (AlarmManager)
            context.getSystemService(Context.ALARM_SERVICE);
      if (isOn) {
         alarmManager.setRepeating(AlarmManager.RTC,
               System.currentTimeMillis(), POLL_INTERVAL, pi);
      } else {
         alarmManager.cancel(pi);
         pi.cancel();
      }
      PreferenceManager.getDefaultSharedPreferences(context)
            .edit()
            .putBoolean(PollService.PREF_IS_ALARM_ON, isOn)
            .commit();
   }
```

Waking Up on a Boot 4/4

Start alarm on boot (StartupReceiver.java)

```
@Override
public void onReceive(Context context, Intent intent) {
    Log.i(TAG, "Received broadcast intent: " + intent.getAction());
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(context);
    boolean isOn = prefs.getBoolean(PollService.PREF_IS_ALARM_ON, false);
    PollService.setServiceAlarm(context, isOn);
}
```

Run PhotoGallery. This time, your background polling should be restarted after you reboot your phone, tablet, or emulator.

Filtering Foreground Notifications

Your notifications work great, but they are sent even when the user already has the application open.

We will implement a logic to send our own broadcast and receive the broadcast with a receiver that can filter out the **notifcation** when **photogallery** is in foreground.

Add the code in PollService.java.

.

```
public class PollService extends IntentService {
    private static final String TAG = "PollService";
    private static final int POLL_INTERVAL = 1000 * 60 * 5; // 5 minutes
    public static final String PREF_IS_ALARM_ON = "isAlarmOn";
    public static final String ACTION_SHOW_NOTIFICATION =
        "com.bignerdranch.android.photogallery.SHOW_NOTIFICATION";
    …
    @Override
    public void onHandleIntent(Intent intent) {
        ...
        if (!resultId.equals(lastResultId)) {
            ...
            NotificationManager notificationManager = (NotificationManager)
                getSystemService(NOTIFICATION_SERVICE);
            notificationManager.notify(0, notification);

            sendBroadcast(new Intent(ACTION_SHOW_NOTIFICATION));
        }
        prefs.edit()
            .putString(FlickrFetchr.PREF_LAST_RESULT_ID, resultId)
            .commit();
    }
```

Filtering Foreground Notifications 1/3

Next up is to receive your broadcast intent. You could write a broadcast receiver like **StartupReceiver** registered in the manifest to handle this.

But that would not do the right thing in your case. Here, you really want **PhotoGalleryFragment** to receive the intent only while it is alive.

A standalone receiver declared in the manifest would not do that job easily. It would always receive the intent, and would need some other way of knowing that **PhotoGalleryFragment** is alive.

The solution is to use a dynamic broadcast receiver. A dynamic receiver is registered in code, not in the manifest.

You register the receiver by calling **registerReceiver(BroadcastReceiver, IntentFilter)** and unregister it by calling **unregisterReceiver(BroadcastReceiver)**.

Filtering Foreground Notifications 2/3

Create a new abstract class called **VisibleFragment**, with **Fragment** as its superclass. This class will be a generic fragment that hides foreground notifications.

```
public abstract class VisibleFragment extends Fragment {
   public static final String TAG = "VisibleFragment";
   private BroadcastReceiver mOnShowNotification = new BroadcastReceiver() {
      @Override
      public void onReceive(Context context, Intent intent) {
         Toast.makeText(getActivity(),
               "Got a broadcast:" + intent.getAction(),
               Toast.LENGTH_LONG)
            .show();
      }
   };
   @Override
   public void onResume() {
      super.onResume();
      IntentFilter filter =
                  new IntentFilter(PollService.ACTION_SHOW_NOTIFICATION);
      getActivity().registerReceiver(mOnShowNotification, filter);
   }
@Override
   public void onPause() {
      super.onPause();
      getActivity().unregisterReceiver(mOnShowNotification);
   }
}
```

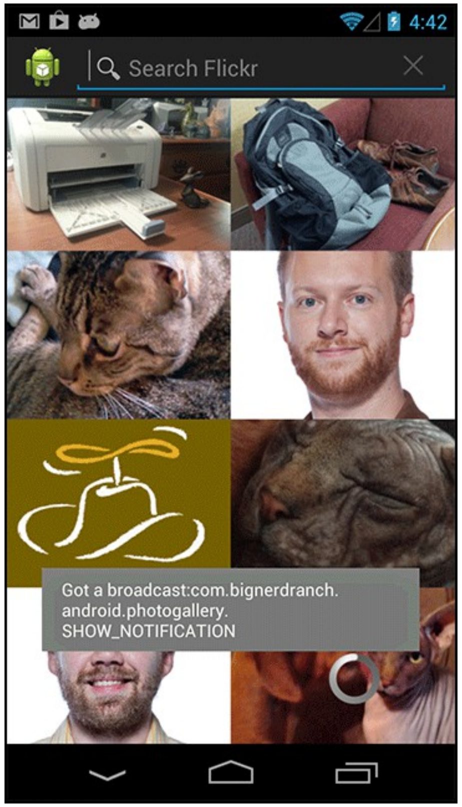
Filtering Foreground Notifications 3/3

Make your fragment visible (PhotoGalleryFragment.java)

```
~~public class PhotoGalleryFragment extends Fragment {~~
public class PhotoGalleryFragment extends VisibleFragment {
    GridView mGridView;
    ArrayList<GalleryItem> mItems;
    ThumbnailDownloader<ImageView> mThumbnailThread;
```

Run PhotoGallery and toggle background polling a couple of times. You will see a nice toast pop up in addition to your notification ticker up top.

Proof that your broadcast exists



```
Got a broadcast:com.bignerdranch.
android.photogallery.
SHOW_NOTIFICATION
```

Using Private Permissions

One problem of broadcast like this is that anyone can listen to it or trigger your receivers.

If the receiver is declared in your manifest and is internal add `android:exported="false"` attribute in receiver tag. This will prevent it from being visible to other .

In other circumstances, you can create your own permission. This is done by adding a `permission` tag to your `AndroidManifest.xml`.

The *signature* protection level means if other applications want to use your permission, it must be signed with the same key as your application.

Add a private permission (AndroidManifest.xml)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.bignerdranch.android.photogallery"
 android:versionCode="1"
 android:versionName="1.0" >
 <uses-sdk
  android:minSdkVersion="8"
  android:targetSdkVersion="17" />
 <permission android:name=
          "com.bignerdranch.android.photogallery.PRIVATE"
           android:protectionLevel="signature" />
 <uses-permission android:name="android.permission.INTERNET" />
 <uses-permission android:name=
          "android.permission.ACCESS_NETWORK_STATE" />
 <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLE
TED" />
 <uses-permission android:name=
           "com.bignerdranch.android.photogallery.PRIVATE" />
 <application
  ... >
  ...
 </application></manifest>
```

Sending with Permission 1/2

(PollService.java)

```
public class PollService extends IntentService {
    private static final String TAG = "PollService";
    private static final int POLL_INTERVAL = 1000 * 60 * 5; // 5 minutes
    public static final String PREF_IS_ALARM_ON = "isAlarmOn";
    public static final String ACTION_SHOW_NOTIFICATION = "com.bignerdranch.android.photogallery.SHOW_NOTIFICATION";

    public static final String PERM_PRIVATE = "com.bignerdranch.android.photogallery.PRIVATE";
     ...
    @Override
    public void onHandleIntent(Intent intent) {
        ...
        if (!resultId.equals(lastResultId)) {
            ...
            NotificationManager notificationManager = (NotificationManager)
                getSystemService(NOTIFICATION_SERVICE);
            notificationManager.notify(0, notification);
            ~~sendBroadcast(new Intent(ACTION_SHOW_NOTIFICATION));~~
             sendBroadcast(new Intent(ACTION_SHOW_NOTIFICATION), PERM_PRIVATE);
        }
      prefs.edit() putString(FlickrFetchr.PREF_LAST_RESULT_ID, resultId) .commit();
    }
```

Sending with Permission 2/2

Permissions on a broadcast receiver (VisibleFragment.java)

```
@Override
   public void onResume() {
      super.onResume();
      IntentFilter filter = new IntentFilter(PollService.ACTION_SHOW_NOTIFICATION);
      ~~getActivity().registerReceiver(mOnShowNotification, filter);~~
       getActivity().registerReceiver(mOnShowNotification, filter,
           PollService.PERM_PRIVATE, null);
   }
```

Now, your app is the only app that can trigger that receiver.

Receiving Results with Ordered Broadcasts 1/2

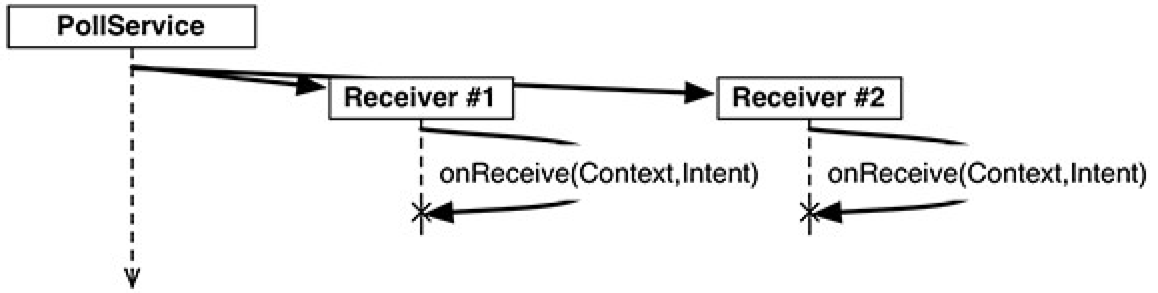
You are sending your own private broadcast, but so far you only have one-way communication.

You can implement two-way communication using an ordered broadcast intent.

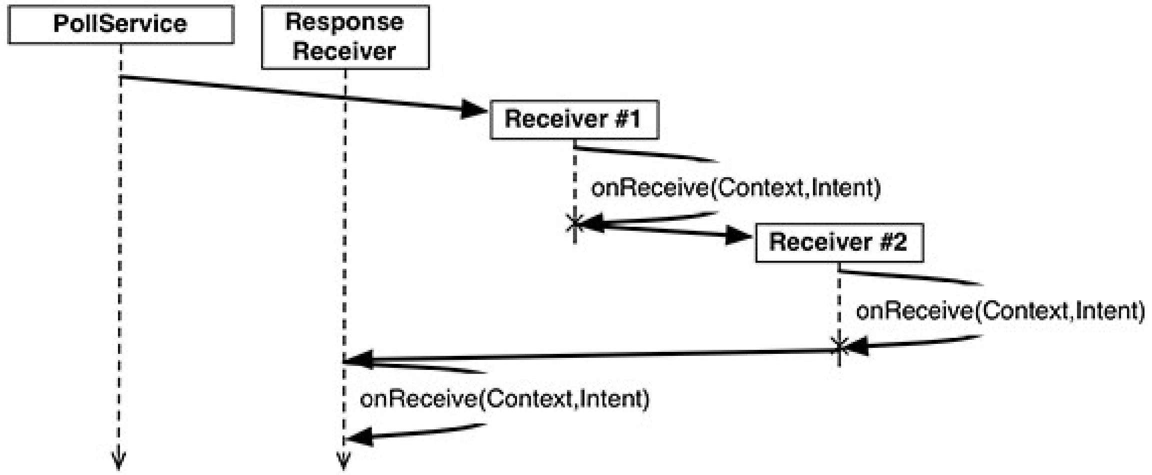
Ordered broadcasts allow a sequence of broadcast receivers to process a broadcast intent in order.

They also allow the sender of a broadcast to receive results from the broadcast's recipients by passing in a special broadcast receiver, called the *result receiver.*

Regular broadcast intents



Ordered broadcast intents


PollService
Response Receiver
Receiver #1
onReceive(Context,Intent)
Receiver #2
onReceive(Context,Intent)
onReceive(Context,Intent)

Receiving Results with Ordered Broadcasts 2/2

On the receiving side, this looks mostly the same as a regular broadcast. You get an additional tool, though: a set of methods used to change the return value of your receiver.

Here, you want to cancel the notification. This can be communicated by use of a simple integer result code. So you use the **setResultCode(int)** method to set the result code to Activity.RESULT_CANCELED.

Send a simple result back (VisibleFragment.java)

```
private BroadcastReceiver mOnShowNotification = new BroadcastReceiver() {
  @Override
  public void onReceive(Context context, Intent intent) {
    ~~Toast.makeText(getActivity(),~~
        ~~"Got a broadcast:" + intent.getAction(),~~
        ~~Toast.LENGTH_LONG)~~
      ~~.show();~~
    // If we receive this, we're visible, so cancel
    // the notification
    Log.i(TAG, "canceling notification");
    setResultCode(Activity.RESULT_CANCELED);
  }
};
```

Send an Ordered Broadcast

(PollService.java)

```
void showBackgroundNotification(int requestCode, Notification notification) {
    Intent i = new Intent(ACTION_SHOW_NOTIFICATION);
    i.putExtra("REQUEST_CODE", requestCode);
    i.putExtra("NOTIFICATION", notification);

    sendOrderedBroadcast(i, PERM_PRIVATE, null, null,
        Activity.RESULT_OK, null, null);
}
```

Context.sendOrderedBroadcast(Intent,String,BroadcastReceiver,Handler,int,String,Bundle) has five additional parameters compare to sendBroadcast(Intent,String). They are, in order: a result receiver, a Handler to run the result receiver on, and then initial values for the result code, result data, and result extras for the ordered broadcast.

The result receiver is a special receiver that will run after all the other recipients of your ordered broadcast intent. In other circumstances, you would be able to use the result receiver here to receive the broadcast and post the notification object. Here, though, that will not work. This broadcast intent will often be sent right before **PollService** dies. That means that your broadcast receiver might be dead, too.

Implement Your Result Receiver

Thus, your final broadcast receiver will need to be standalone. Create a new **BroadcastReceiver** subclass called **NotificationReceiver**.

```
public class NotificationReceiver extends BroadcastReceiver {
    private static final String TAG = "NotificationReceiver";

    @Override
    public void onReceive(Context c, Intent i) {
        Log.i(TAG, "received result: " + getResultCode());
        if (getResultCode() != Activity.RESULT_OK)
            // A foreground activity cancelled the broadcast
            return;

        int requestCode = i.getIntExtra("REQUEST_CODE", 0);
        Notification notification = (Notification)
                i.getParcelableExtra("NOTIFICATION");

        NotificationManager notificationManager = (NotificationManager)
                c.getSystemService(Context.NOTIFICATION_SERVICE);
        notificationManager.notify(requestCode, notification);
    }
}
```

Register Notification Receiver

Finally, register your new receiver. Since it sends the notification, receiving the result set by the other receivers, it should run after everything else.

That means that you will need to set a low priority for your receiver. Since your receiver should run last, give it a priority of -999 (-1000 and below are reserved).

And since this receiver is only used internally by your application, you do not need it to be externally visible. Set `android:exported="false"` to keep this receiver to yourself.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 ... >
 ...
 <application
  ... >
  ...
  <receiver android:name=".StartupReceiver">
   <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
   </intent-filter>
  </receiver>
  <receiver android:name=".NotificationReceiver"
    android:exported="false">
    <intent-filter
      android:priority="-999">
      <action android:name=
       "com.bignerdranch.android.photogallery.SHOW_NOTIFICATION" />
    </intent-filter>
  </receiver>
 </application></manifest>
```

Finish It Up

Now use your new method instead of **NotificationManager** to post your notification.

(PollService.java)

```
@Override
public void onHandleIntent(Intent intent) {
    ...
    if (!resultId.equals(lastResultId)) {
        ...
        Notification notification = new NotificationCompat.Builder(this)
            ...
            .build();
        ~~NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);~~
        ~~notificationManager.notify(0, notification);~~
        ~~sendBroadcast(new Intent(ACTION_SHOW_NOTIFICATION), PERM_PRIVATE);~~
        showBackgroundNotification(0, notification);
    }
    prefs.edit()
        .putString(FlickrFetchr.PREF_LAST_RESULT_ID, resultId)
        .commit();
}
```

Android Networking
Browsing the Web and WebView

Javed Hasan
BJIT Limited

Browsing the Web and WebView

PhotoGallery Application New Feature

- Browse Photo Page using Web Browser and WebView

Browse Photo Page using Web Browser

- Web Browsing with Implicit Intent

Browse Photo Page with WebView

- Add Progress Bar and Title Text using WebChromeClient
- Proper Rotation with WebView

Flickr Data for Photo Page Retrieval

Each photo you get from Flickr has a page associated with it. You are going to make it so that users can press a photo in PhotoGallery to browse its photo page.

Flickr's documentation at http://www.flickr.com/services/api/misc.urls.html says, you can create the URL for an individual photo's page like so with:

```
http://www.flickr.com/photos/user-id/photo-id
```

If you look at the XML you are currently receiving for each photo, you can see that there is an owner attribute. This owner is the user-id of the above url.

```
<photo id="8232706407" owner="70490293@N03" secret="9662732625"
   server="8343" farm="9" title="111_8Q1B2033" ispublic="1"
   isfriend="0" isfamily="0"
   url_s="http://farm9.staticflickr.com/8343/8232706407_9662732625_m.jpg"
   height_s="240" width_s="163" />
```

The photo-id seen here is the same as the value of the id attribute from your XML. We are already saving the photo-id of each photo in photogallery app.

Add Code for Photo Page

(GalleryItem.java)

```
public class GalleryItem {
   private String mCaption;
   private String mId;
   private String mUrl;
   private String mOwner;
   ...
   public void setUrl(String url) {
      mUrl = url;
   }
   public String getOwner() {
        return mOwner;
    }
    public void setOwner(String owner) {
        mOwner = owner;
    }
    public String getPhotoPageUrl() {
        return "http://www.flickr.com/photos/"
+ mOwner + "/" + mId;
    }
   public String toString() {
      return mCaption;
   }}
```

Read in owner attribute (FlickrFetchr.java)

```
void parseItems(ArrayList<GalleryItem> items, XmlPullParser parser)
      throws XmlPullParserException, IOException {
   int eventType = parser.next();
   while (eventType != XmlPullParser.END_DOCUMENT) {
      if (eventType == XmlPullParser.START_TAG &&
         XML_PHOTO.equals(parser.getName())) {
         String id = parser.getAttributeValue(null, "id");
         String caption = parser.getAttributeValue(null, "title");
         String smallUrl = parser.getAttributeValue(null, EXTRA_SMALL_URL);
         String owner = parser.getAttributeValue(null, "owner");
         GalleryItem item = new GalleryItem();
         item.setUrl(smallUrl);
         item.setOwner(owner);
         items.add(item);
      }
      eventType = parser.next();
   }
}
```

Web Browsing with Implicit Intents

This Implicit intent will start up the browser with your photo URL. The first step here is to make your app listen to presses on GridView items. You will do it up by calling the setOnItemClickListener(…) method on your GridView. The you will create and fire an implicit intent in the listener code.

(PhotoGalleryFragment.java)

```
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_photo_gallery, container, false);
    mGridView = (GridView)v.findViewById(R.id.gridView);
    setupAdapter();
    mGridView.setOnItemClickListener(new OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> gridView, View view, int pos,
                long id) {
            GalleryItem item = mItems.get(pos);
            Uri photoPageUri = Uri.parse(item.getPhotoPageUrl());
            Intent i = new Intent(Intent.ACTION_VIEW, photoPageUri);
            startActivity(i);
        }
    });
    return v;
}
```

Web Browsing with WebView

Oftentimes, though, you want to display web content within your own activities instead of heading off to the browser.

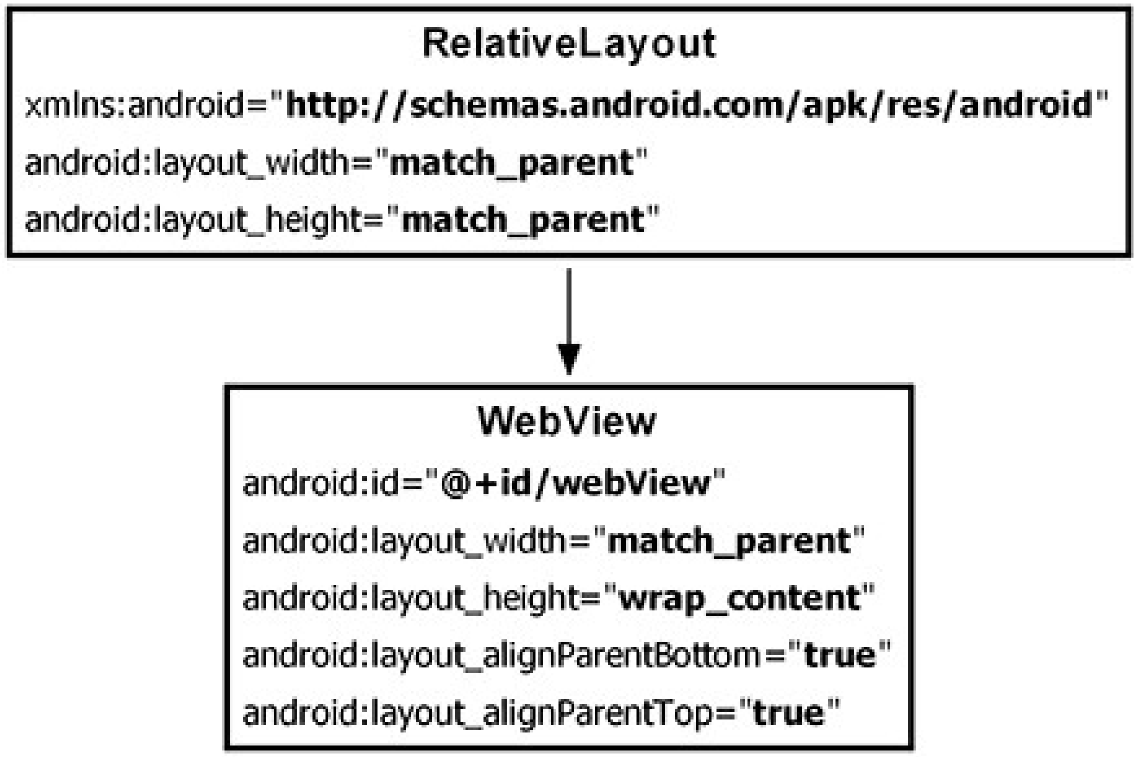
You may want to display HTML that you generate yourself, or you may want to lock down the browser somehow.

For apps that include help documentation, it is common to implement it as a web page so that it is easy to update. Opening a web browser to a help web page does not look professional, though, and it prevents you from customizing behavior or integrating that web page into your own user interface.

The first step will be to create a new activity and fragment to display the **WebView** in.

Initial Layout for Showing Photo Page

(res/layout/fragment_photo_page.xml)


RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent"
WebView
android:id="@+id/webView"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true"
android:layout_alignParentTop="true"

Setting Up Your Web Browser Fragment

(PhotoPageFragment.java)

```
public class PhotoPageFragment extends VisibleFragment {
    private String mUrl;
    private WebView mWebView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        mUrl = getActivity().getIntent().getData().toString();
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup parent,
            Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_photo_page, parent, false);
          mWebView = (WebView)v.findViewById(R.id.webView);
        return v;
    }
}
```

Create Web Activity

(PhotoPageActivity.java)

```
public class PhotoPageActivity
        extends SingleFragmentActivity {
   @Override
   public Fragment createFragment() {
      return new PhotoPageFragment();
   }
}
```

Add activity to manifest (AndroidManifest.xml)

```
<manifest ... >
  ...
  <application
    ... >
    <activity
      android:name=".PhotoGalleryActivity"
      android:launchMode="singleTop"
      android:label="@string/title_activity_photo_gallery" >
      ...
    </activity>
    <activity android:name=".PhotoPageActivity" />
    <service android:name=".PollService" />
    <receiver android:name=".StartupReceiver">
       ...
    </receiver>
  </application>
</manifest>
```

Switch to Call Your Activity

(PhotoGalleryFragment.java)

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
      Bundle savedInstanceState) {
   ...
   mGridView.setOnItemClickListener(new OnItemClickListener() {
      @Override
      public void onItemClick(AdapterView<?> gridView, View view, int pos,
            long id) {
         GalleryItem item = mItems.get(pos);
         Uri photoPageUri = Uri.parse(item.getPhotoPageUrl());
         ~~Intent i = new Intent(Intent.ACTION_VIEW, photoPageUri);~~
         Intent i = new Intent(getActivity(), PhotoPageActivity.class);
         i.setData(photoPageUri);
         startActivity(i);
      }
   });

   return v;
}
```

Run PhotoGallery and press on a picture. You should see a new empty activity pop up.

Loading and Viewing Web Pages 1/3

You need to do three things to make your **WebView** successfully display a Flickr photo page.

The **first** one is straightforward – you need to tell it what URL to load.

The **second** thing you need to do is enable JavaScript. By default, JavaScript is off. You do not always need to have it on, but for Flickr, you do.

Android Lint gives you a warning for doing this (it is worried about cross-site scripting attacks), so you also need to suppress Lint warnings.

Finally, you need to override one method on a class called **WebViewClient, shouldOverrideUrlLoading(WebView,String),** and return false.

We will discuss this class a bit more after you type in the code.

Loading and Viewing Web Pages 2/3

(PhotoPageFragment.java)

```
@SuppressLint("SetJavaScriptEnabled")
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent,
        Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.fragment_photo_page, parent, false);
    mWebView = (WebView)v.findViewById(R.id.webView);
    mWebView.getSettings().setJavaScriptEnabled(true);
     mWebView.setWebViewClient(new WebViewClient() {
          public boolean shouldOverrideUrlLoading(WebView view, String url) {
               return false;
          }
     });
     mWebView.loadUrl(mUrl);
    return v;
}
```

Loading and Viewing Web Pages 3/3

WebViewClient is an event interface. By providing your own implementation of **WebViewClient**, you can respond to rendering events. For example, you could detect when the renderer starts loading an image from a particular URL.

In shouldOverrideUrlLoading(WebView,String), If you return true, you are saying, “Do not handle this URL, I am handling it myself.” If you return false, you are saying, “Go ahead and load this URL, **WebView**, I’m not doing anything with it.”

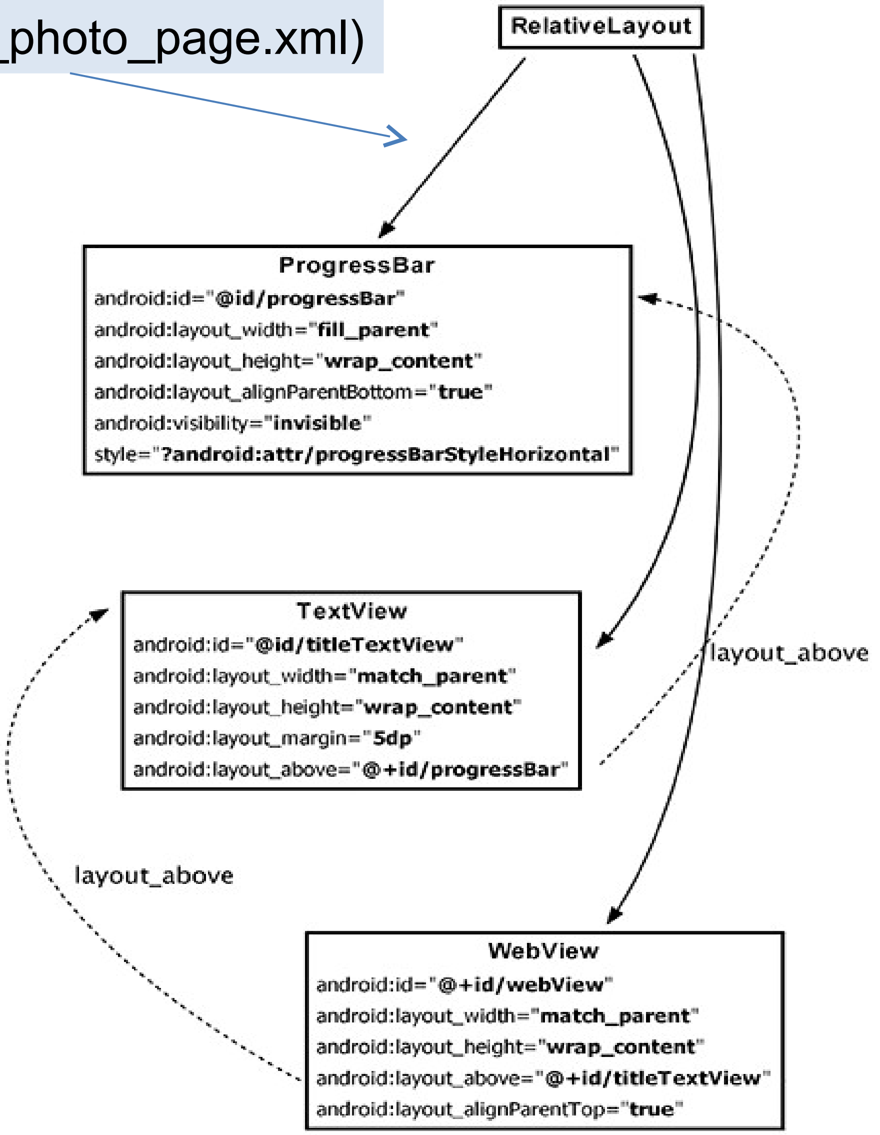
The default implementation fires an implicit intent with the URL. For your photo page, this is a severe problem. The first thing Flickr does is redirect you to the mobile version of the web site. With the default **WebViewClient**, that means that you are immediately sent to the user’s default web browser. Not ideal.

The fix is simple – just override the default implementation and return false.

Run PhotoGallery, and you should see your **WebView**.

Using WebChromeClient to enhance the UI with Progress Bar and Title 1/2

(fragment_photo_page.xml)



ProgressBar
android:id="@id/progressBar"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:layout_alignParentBottom="true"
android:visibility="invisible"
style="?android:attr/progressBarStyleHorizontal"

TextView
android:id="@id/titleTextView"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_margin="5dp"
android:layout_above="@+id/progressBar"

WebView
android:id="@+id/webView"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:layout_above="@+id/titleTextView"
android:layout_alignParentTop="true"

Using WebChromeClient to enhance the UI with Progress Bar and Title 2/2

```
@SuppressLint("SetJavaScriptEnabled")
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup parent,
       Bundle savedInstanceState) {
   View v = inflater.inflate(R.layout.fragment_photo_page, parent, false);
   final ProgressBar progressBar = (ProgressBar)v.findViewById(R.id.progress
Bar);
    progressBar.setMax(100); // WebChromeClient reports in range 0-100
    final TextView titleTextView = (TextView)v.findViewById(R.id.titleTextView);
   mWebView = (WebView)v.findViewById(R.id.webView);
   mWebView.getSettings().setJavaScriptEnabled(true);
   mWebView.setWebViewClient(new WebViewClient() {
      ...
   });
mWebView.setWebChromeClient(new WebChromeClient() {
        public void onProgressChanged(WebView webView, int progress) {
            if (progress == 100) {
                progressBar.setVisibility(View.INVISIBLE);
            } else {
                progressBar.setVisibility(View.VISIBLE);
                progressBar.setProgress(progress);
            }
        }
        public void onReceivedTitle(WebView webView,
                String title) {
            titleTextView.setText(title);
        }
    });
    mWebView.loadUrl(mUrl);
    return v;
}
```

The progress you receive from **onProgressChanged(WebView,int)** is an integer from 0 to 100. If it is 100, you know that the page is done loading, so you hide the ProgressBar by setting its visibility to View.INVISIBLE.

Use **onReceivedTitle(WebView,String)** to update title of the photo.

Proper Rotation with WebView

Try rotating your screen. While it does work correctly, you will notice that the **WebView** has to completely reload the web page.

For some classes like this (**VideoView** is another one), the Android documentation recommends that you allow the activity to handle the configuration change itself.

This means that instead of the activity being destroyed, it simply moves its views around to fit the new screen size.

As a result, **WebView** does not have to reload all of its data.

```
<manifest ... >
 ...
 <application
  android:allowBackup="true"
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme" >
  ...
  <activity
     android:name=".PhotoPageActivity"
     android:configChanges="keyboardHidden|orientation|screenSize" />
  ...
 </application>
</manifest>
```

This attribute says that if the configuration changes because the keyboard was opened or closed, due to an orientation change or due to the screen size changing then the activity should handle the change itself.

Android Networking
Tracking the Device’s Location

Javed Hasan

BJIT Limited

Tracking the Device's Location

RunTracker Application Features

- Getting Started with RunTracker App

Locations and the LocationManager

- RunManager Class to Keep Track of Each Run
- Receiving Broadcast Location Updates

UI Update with Location Data

- Faster Answers: the Last Known Location

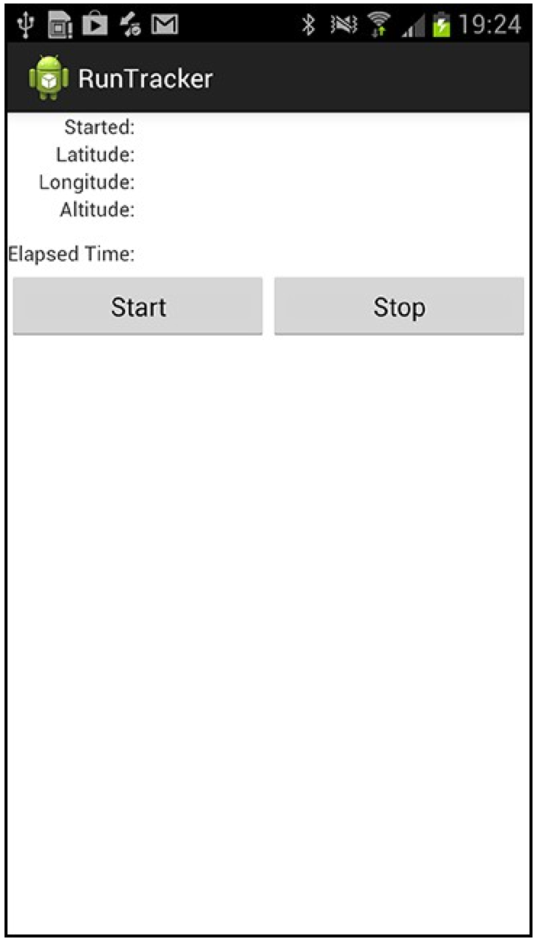
Testing Locations on Real and Virtual Devices

RunTracker App Overview

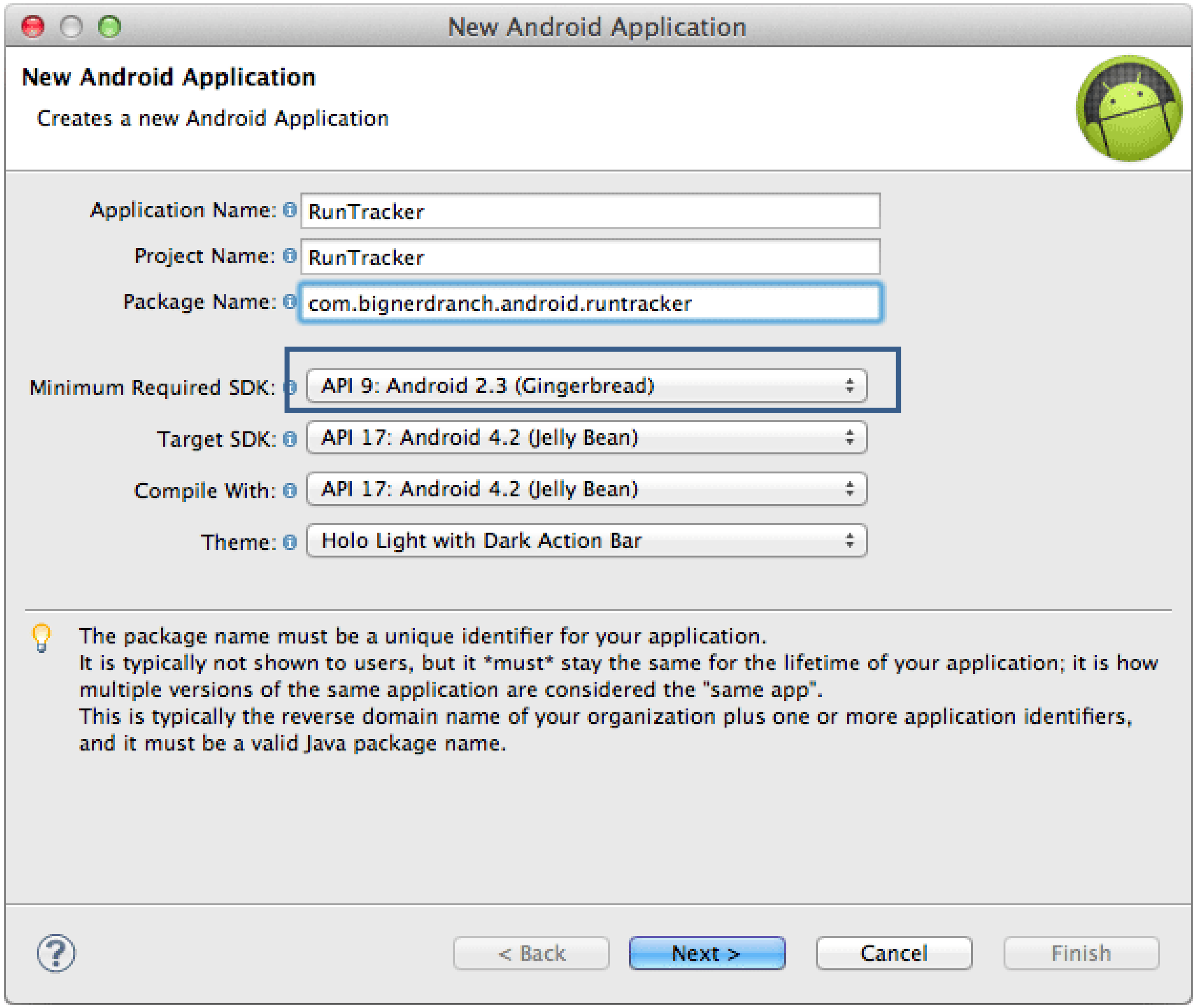
RunTracker works with a device’s GPS to record and display the user’s travels.

Those travels may be a walk in the woods, a car ride, or an ocean voyage. RunTracker is designed to keep a record of them all.

RunTracker will simply get location updates from the GPS and display the device’s current location on screen.



Creating RunTracker



New Android Application

New Android Application

Creates a new Android Application

Application Name: RunTracker

Project Name: RunTracker

Package Name: com.bignerdranch.android.runtracker

Minimum Required SDK: API 9: Android 2.3 (Gingerbread)

Target SDK: API 17: Android 4.2 (Jelly Bean)

Compile With: API 17: Android 4.2 (Jelly Bean)

Theme: Holo Light with Dark Action Bar

The package name must be a unique identifier for your application.
It is typically not shown to users, but it *must* stay the same for the lifetime of your application; it is how multiple versions of the same application are considered the "same app".
This is typically the reverse domain name of your organization plus one or more application identifiers, and it must be a valid Java package name.

Setting up RunActivity & RunFragment 1/2

Initial RunActivity (RunActivity.java)

```
public class RunActivity extends SingleFragmentActivity {
   @Override
   protected Fragment createFragment() {
      return new RunFragment();
   }
}
```

RunTracker strings (strings.xml)

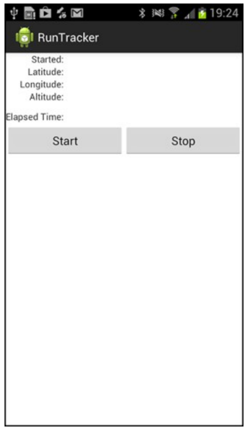
```
<resources>
  <string name="app_name">RunTracker</string>
  <string name="started">Started:</string>
  <string name="latitude">Latitude:</string>
  <string name="longitude">Longitude:</string>
  <string name="altitude">Altitude:</string>
  <string name="elapsed_time">Elapsed Time:</string>
  <string name="start">Start</string>
  <string name="stop">Stop</string>
  <string name="gps_enabled">GPS Enabled</string>
  <string name="gps_disabled">GPS Disabled</string>
  <string name="cell_text">Run at %1$s</string>
</resources>
```

Obtaining the layout file

For the layout, you will use a **TableLayout** to keep everything tidy. The TableLayout will have five **TableRows** and one **LinearLayout**.

Each **TableRow** will have two **TextViews**: the first is a label, and the second will be populated with data at runtime. The **LinearLayout** will contain two **Buttons**.

Find 33_Location/RunTracker/res/layout/fragment_run.xml and copy it into your project's res/layout directory.



Setting up RunActivity & RunFragment 2/2

```
public class RunFragment extends Fragment {
   private Button mStartButton, mStopButton;
   private TextView mStartedTextView, mLatitudeTextView,
      mLongitudeTextView, mAltitudeTextView, mDurationTextView;
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setRetainInstance(true);
   }
   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
         Bundle savedInstanceState) {
      View view = inflater.inflate(R.layout.fragment_run, container, false);
      mStartedTextView = (TextView)view.findViewById(R.id.run_startedTextView);
      mLatitudeTextView = (TextView)view.findViewById(R.id.run_latitudeTextView);
      mLongitudeTextView = (TextView)view.findViewById(R.id.run_longitudeTextView);
      mAltitudeTextView = (TextView)view.findViewById(R.id.run_altitudeTextView);
      mDurationTextView = (TextView)view.findViewById(R.id.run_durationTextView);
      mStartButton = (Button)view.findViewById(R.id.run_startButton);
      mStopButton = (Button)view.findViewById(R.id.run_stopButton);
      return view;
   }
}
```

Locations and the LocationManager

LocationManager system service provides location updates to all applications who are interested in them, and it delivers those updates in one of two ways.

One way is the **LocationListener** interface. This interface gives you information about location updates (via **onLocationChanged(Location)**) as well as status updates and notifications that the provider has been enabled or disabled.

Using **LocationListener** is a fine thing when you only need the location data to come to a single component in your application. For example, if you only wanted to display location updates within **RunFragment**. You could provide a **LocationListener** implementation to a call to **LocationManager's** request **LocationUpdates(…)** and be done.

Second way is using Pending Intent. By requesting location updates with a **PendingIntent**, you are asking the **LocationManager** to send some kind of Intent for you in the future. So, your application components, and indeed your entire process, can die, and **LocationManager** will still deliver your intents until you tell it to stop, starting new components to respond to them as needed.

Locations and the LocationManager

LocationManager system service provides location updates to all applications who are interested in them, and it delivers those updates in one of two ways.

One way is the **LocationListener** interface. This interface gives you information about location updates (via **onLocationChanged(Location)**) as well as status updates and notifications that the provider has been enabled or disabled.

Using **LocationListener** is a fine thing when you only need the location data to come to a single component in your application. For example, if you only wanted to display location updates within **RunFragment**. You could provide a **LocationListener** implementation to a call to **LocationManager's** request **LocationUpdates(…)** and be done.

Second way is using Pending Intent. By requesting location updates with a **PendingIntent**, you are asking the **LocationManager** to send some kind of Intent for you in the future. So, your application components, and indeed your entire process, can die, and **LocationManager** will still deliver your intents until you tell it to stop, starting new components to respond to them as needed.

RunManager Class 1/2

```
public class RunManager {
    private static final String TAG = "RunManager";
    public static final String ACTION_LOCATION =
        "com.bignerdranch.android.runtracker.ACTION_LOCATION";
    private static RunManager sRunManager;
    private Context mAppContext;
    private LocationManager mLocationManager;
    // The private constructor forces users to use RunManager.get(Context)
    private RunManager(Context appContext) {
        mAppContext = appContext;
        mLocationManager = (LocationManager)mAppContext
            .getSystemService(Context.LOCATION_SERVICE);
    }
    public static RunManager get(Context c) {
        if (sRunManager == null) {
            // Use the application context to avoid leaking activities
            sRunManager = new RunManager(c.getApplicationContext());
        }
        return sRunManager;
    }
```

To manage the communication with **LocationManager**, and eventually more details about the current run, create a singleton class called **RunManager.**

RunManager has three public instance methods. These are its basic API. It can start location updates, stop them, and tell you if it is currently tracking a run (which merely means that updates are currently registered with the **LocationManager**).

RunManager Class 2/2

```
private PendingIntent getLocationPendingIntent(boolean shouldCreate) {
    Intent broadcast = new Intent(ACTION_LOCATION);
    int flags = shouldCreate ? 0 : PendingIntent.FLAG_NO_CREATE;
    return PendingIntent.getBroadcast(mAppContext, 0, broadcast, flags);
  }
  public void startLocationUpdates() {
    String provider = LocationManager.GPS_PROVIDER;
    // Start updates from the location manager
    PendingIntent pi = getLocationPendingIntent(true);
    mLocationManager.requestLocationUpdates(provider, 0, 0, pi);
  }
  public void stopLocationUpdates() {
    PendingIntent pi = getLocationPendingIntent(false);
    if (pi != null) {
      mLocationManager.removeUpdates(pi);
      pi.cancel();
    }
  }
  public boolean isTrackingRun() {
    return getLocationPendingIntent(false) != null;
  }
}
```

In **startLocationUpdates()**, you specifically tell **LocationManager** to give you location updates via the GPS provider as frequently as possible.

The **requestLocationUpdates(String, long, float, PendingIntent)** method expects parameters for the minimum time to wait (in milliseconds) and minimum distance to cover (in meters) before sending the next update.

The private **getLocationPendingIntent(boolean)** method creates an Intent to be broadcast when location updates happen.

Receiving Broadcast Location Updates

```
Basic LocationReceiver (LocationReceiver.java)
public class LocationReceiver extends BroadcastReceiver {
  private static final String TAG = "LocationReceiver";
  @Override
  public void onReceive(Context context, Intent intent) {
      // If you got a Location extra, use it
      Location loc = (Location)intent
          .getParcelableExtra(LocationManager.KEY_LOCATION_CHANGED);
      if (loc != null) {
          onLocationReceived(context, loc);
          return;
      }
      // If you get here, something else has happened
      if (intent.hasExtra(LocationManager.KEY_PROVIDER_ENABLED)) {
          boolean enabled = intent
              .getBooleanExtra(LocationManager.KEY_PROVIDER_ENABLED, false);
          onProviderEnabledChanged(enabled);
      }
  }
  protected void onLocationReceived(Context context, Location loc) {
      Log.d(TAG, this + " Got location from " + loc.getProvider() + ": "
          + loc.getLatitude() + ", " + loc.getLongitude());
  }
protected void onProviderEnabledChanged(boolean enabled) {
     Log.d(TAG, "Provider " + (enabled ? "enabled" : "disabled"));
  }
}
```

RunTracker will need to be able to receive them whether or not any UI components, or even the application process, are running, so the best place to handle it is with a standalone **BroadcastReceiver** registered in the manifest.

Adding Location Permission in Manifest

```
<manifest...>
  <uses-sdk android:minSdkVersion="9" android:targetSdkVersion="15" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-feature android:required="true"
    android:name="android.hardware.location.gps"/>

  <application ...>
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <receiver android:name=".LocationReceiver"
       android:exported="false">
       <intent-filter>
         <action android:name="com.bignerdranch.android.runtracker.ACTION_LOCATION"/>
       </intent-filter>
     </receiver>
  </application>
</manifest>
```

Updating the UI with Location Data 1/5

Starting and stopping location updates (RunFragment.java)

```
public class RunFragment extends Fragment {
   private RunManager mRunManager;
   private Button mStartButton, mStopButton;
   private TextView mStartedTextView, mLatitudeTextView,
      mLongitudeTextView, mAltitudeTextView, mDurationTextView;
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setRetainInstance(true);
      mRunManager = RunManager.get(getActivity());
   }
   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
         Bundle savedInstanceState) {
      ...
      mStartButton = (Button)view.findViewById(R.id.run_startButton);
      mStartButton.setOnClickListener(new View.OnClickListener() {
           @Override
           public void onClick(View v) {
               mRunManager.startLocationUpdates();
               updateUI();
           }
       });

      mStopButton = (Button)view.findViewById(R.id.run_stopButton);
      mStopButton.setOnClickListener(new View.OnClickListener() {
           @Override
           public void onClick(View v) {
               mRunManager.stopLocationUpdates();
               updateUI();
           }
       });
      updateUI();
      return view;
   }
   private void updateUI() {
      boolean started = mRunManager.isTrackingRun();
      mStartButton.setEnabled(!started);
      mStopButton.setEnabled(started);
   }
}
```

You can run **RunTracker** again and see location updates coming in via **LogCat**.

Updating the UI with Location Data 2/5

Your basic Run (Run.java)

```
public class Run {
   private Date mStartDate;
   public Run() {
      mStartDate = new Date();
   }
   public Date getStartDate() {
      return mStartDate;
   }
   public void setStartDate(Date startDate) {
      mStartDate = startDate;
   }
   public int getDurationSeconds(long endMillis) {
      return (int)((endMillis - mStartDate.getTime()) / 1000);
   }
   public static String formatDuration(int durationSeconds) {
      int seconds = durationSeconds % 60;
      int minutes = ((durationSeconds - seconds) / 60) % 60;
      int hours = (durationSeconds - (minutes * 60) - seconds) / 3600;
      return String.format("%02d:%02d:%02d", hours, minutes, seconds);
   }
}
```

Logging to **LogCat** is not a very user-friendly way of reporting locations.

To get something on the screen, you can implement a subclass of **LocationReceiver** in **RunFragment** that will stash away the Location and update the UI.

With a bit more data, stored in a new **Run** instance, you can display the start date and duration of the current run.

Updating the UI with Location Data 3/5

Displaying location updates (RunFragment.java)

```
public class RunFragment extends Fragment {
 private BroadcastReceiver mLocationReceiver = new LocationReceiver() {
     @Override
     protected void onLocationReceived(Context context, Location loc) {
         mLastLocation = loc;
         if (isVisible())
             updateUI();
     }
    @Override
    protected void onProviderEnabledChanged(boolean enabled) {
       int toastText = enabled ? R.string.gps_enabled : R.string.gps_disabled;
       Toast.makeText(getActivity(), toastText, Toast.LENGTH_LONG).show();
    }
 };
 private RunManager mRunManager;
 private Run mRun;
 private Location mLastLocation;
 private Button mStartButton, mStopButton;
 ...
```

```
@Override
 public View onCreateView(LayoutInflater inflater,
       ViewGroup container,
       Bundle savedInstanceState) {
   ...
   mStartButton = (Button)view.findViewById(
         R.id.run_startButton);
   mStartButton.setOnClickListener(
        new View.OnClickListener() {
       @Override
       public void onClick(View v) {
          mRunManager.startLocationUpdates();
          mRun = new Run();
          updateUI();
       }
   });
   ...
 }
```

Updating the UI with Location Data 4/5

```
@Override
  public void onStart() {
      super.onStart();
      getActivity().registerReceiver(mLocationReceiver,
              new IntentFilter(RunManager.ACTION_LOCATION));
  }
  @Override
  public void onStop() {
      getActivity().unregisterReceiver(mLocationReceiver);
      super.onStop();
  }

  private void updateUI() {
     boolean started = mRunManager.isTrackingRun();
     if (mRun != null)
        mStartedTextView.setText(mRun.getStartDate().toString());
     int durationSeconds = 0;
     if (mRun != null && mLastLocation != null) {
        durationSeconds = mRun.getDurationSeconds(mLastLocation.getTime());
        mLatitudeTextView.setText(Double.toString(mLastLocation.getLatitude()));
        mLongitudeTextView.setText(Double.toString(mLastLocation.getLongitude()));
        mAltitudeTextView.setText(Double.toString(mLastLocation.getAltitude()));
     }
     mDurationTextView
        .setText(Run.formatDuration(durationSeconds));
     mStartButton.setEnabled(!started);
     mStopButton.setEnabled(started);
   }
}
```

Updating the UI with Location Data 5/5

There are new instance variables for a **Run** and the last **Location** received. These are the data that back up the user interface updates performed in **updateUI().** The Run gets initialized as soon as you start location updates.

You create an anonymous **LocationReceiver** class and stash it in **mLocationReceiver** to save the location received and update the UI. You also display a **Toast** when the GPS provider is enabled or disabled.

Finally, the implementations of the **onStart()** and **onStop()** methods are used to register and unregister the receiver in conjunction with the fragment being visible to the user.

Faster Answers: the Last Known Location

Getting the last known location (RunManager.java)

```
public void startLocationUpdates() {
    String provider = LocationManager.GPS_PROVIDER;
    // Get the last known location and broadcast it if you have one
     Location lastKnown = mLocationManager.getLastKnownLocation(provider);
     if (lastKnown != null) {
          // Reset the time to now
          lastKnown.setTime(System.currentTimeMillis());
          broadcastLocation(lastKnown);
     }
    // Start updates from the location manager
    PendingIntent pi = getLocationPendingIntent(true);
    mLocationManager.requestLocationUpdates(provider, 0, 0, pi);
}
private void broadcastLocation(Location location) {
     Intent broadcast = new Intent(ACTION_LOCATION);
     broadcast.putExtra(LocationManager.KEY_LOCATION_CHANGED, location);
     mAppContext.sendBroadcast(broadcast);
}
```

It takes sometime to get GPS reading.

Use **LocationManager's** GPS provider to get the last known location reading. Thus you can prevent waiting your user.

The only trick is getting that location back to the user interface, and for that you can simply broadcast an Intent just as though you were the **LocationManager**.

Testing Locations on Real and Virtual Devices 1/2

Testing an app like **RunTracker** can be challenging. You want to ensure that the locations you receive from the system are being appropriately tracked and stored. This can be difficult to do if you are moving around even at low speed.

To get around situations like this, you can send test locations to the **LocationManager** that will allow your device to pretend that it is somewhere else.

The simplest way to make this happen is using the Emulator Control window within DDMS. This only works with virtual devices, but it allows you to specify new locations either manually, one at a time, or with a GPX or KML file representing a series of locations visited over time.

For testing locations on a real device, you have a bit more work to do, but it is entirely possible. The basic process looks like this:

1. Request the ACCESS_MOCK_LOCATION permission.
2. Add a test provider **via LocationManager.addTestProvider(…).**
3. Enable the provider **using setTestProviderEnabled(…).**
4. Set its initial status with **setTestProviderStatus(…).**
5. Publish locations with **setTestProviderLocation(…).**
6. Remove your test provider using **removeTestProvider(…).**

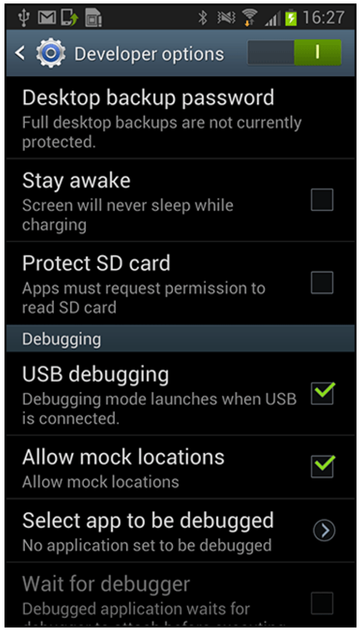
Testing Locations on Real and Virtual Devices 2/2

Using a test provider (RunManager.java)

```
public class RunManager {
    private static final String TAG = "RunManager";
    public static final String ACTION_LOCATION =
        "com.bignerdranch.android.runtracker.ACTION_LOCATION";
    private static final String TEST_PROVIDER = "TEST_PROVIDER";
    ...
    public void startLocationUpdates() {
        String provider = LocationManager.GPS_PROVIDER;
        // If you have the test provider and it's enabled, use it
        if (mLocationManager.getProvider(TEST_PROVIDER) != null &&
                mLocationManager.isProviderEnabled(TEST_PROVIDER)) {
            provider = TEST_PROVIDER;
        }
        Log.d(TAG, "Using provider " + provider);
        // get the last known location and broadcast it if you have one
        Location lastKnown = mLocationManager.getLastKnownLocation(provider);
        if (lastKnown != null) {
            // Reset the time to now
            lastKnown.setTime(System.currentTimeMillis());
            broadcastLocation(lastKnown);
        }
    ...
```

Fortunately, you can use the **TestProvider** project to do all the hard works. Install it to your device, and run to manage the test provider and make it run.

In order for **TestProvider** to work, you may need to turn on the Allow mock locations setting in the Developer options menu from within the Settings application.



Desktop backup password
Full desktop backups are not currently protected.

Stay awake
Screen will never sleep while charging

Protect SD card
Apps must request permission to read SD card

Debugging

USB debugging ✓
Debugging mode launches when USB is connected.

Allow mock locations ✓
Allow mock locations

Select app to be debugged
No application set to be debugged

Wait for debugger
Debugged application waits for

Android Networking
Local Databases with SQLite

Javed Hasan
BJIT Limited

Local Databases with SQLite

RunTracker Application Data Model

- Save Runs and Locations Data in SQLite DB

Querying a List of Runs from DB

- Using CursorWrapper class to Get Run Object from Query

Displaying a List of Runs Using CursorAdapter

- Implementing RunCursorAdapter

Creating New Runs and Working with Existing Runs

Storing Runs and Locations in a Database

In RunTracker, the user can continue tracking his or her location forever, which can generate a lot of data. The logical choice for such data sets in Android is a SQLite database.

Android includes a Java front-end to SQLite through the **SQLiteDatabase** class, which provides result sets as **Cursor** instances. In this lesson, you will create a storage mechanism for RunTracker that uses a database to store data about runs and their locations.

in Android, a helper class **SQLiteOpenHelper** encapsulates the chore of creating, opening, and updating databases for storing your application data.

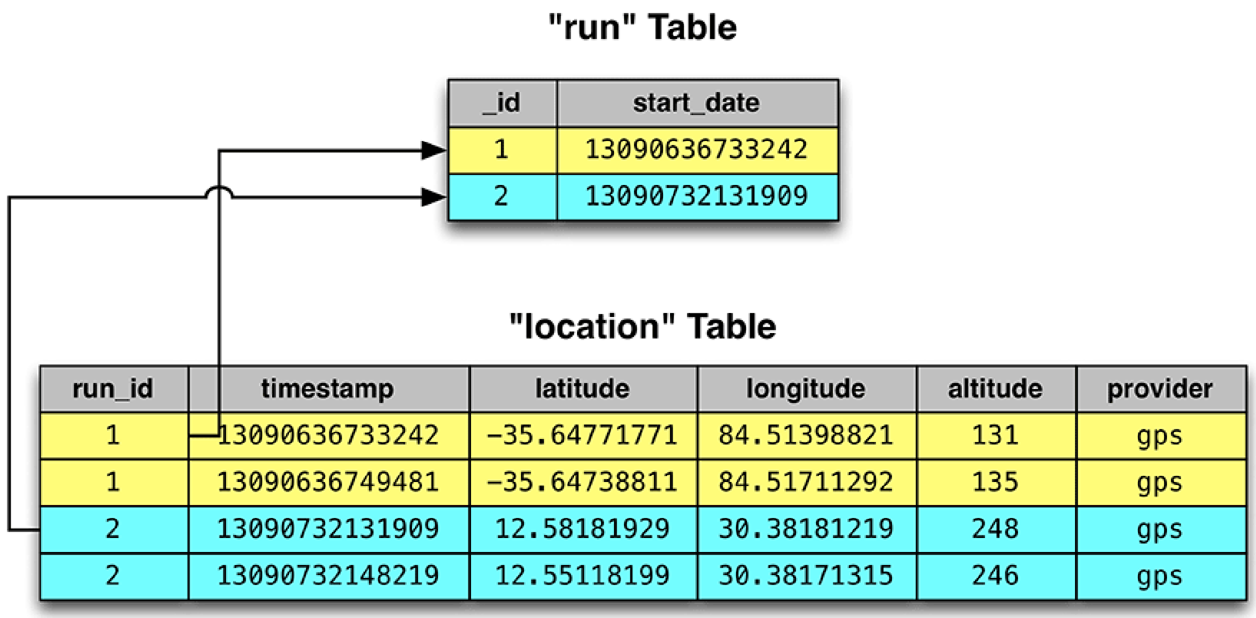
In RunTracker, you will create a subclass of **SQLiteOpenHelper** called **RunDatabaseHelper**. The **RunManager** will hold on to a private instance of **RunDatabaseHelper** and provide the rest of the application with an API for inserting, querying, and otherwise managing the data in the database.

RunDatabaseHelper will provide methods that **RunManager** will call to implement most of its API.

RunTracker App Data Model

In object oriented programming, the most common pattern for database design is to use one database table for each class in your application's data model. For RunTracker, there are two classes to store: **Run** and **Location**.

This example will therefore create two tables: run and location. A **Run** can have many **Locations**, so the location table will have a run_id foreign key column referencing the run table's _id column.



"run" Table

_id	start_date
1	13090636733242
2	13090732131909

"location" Table

run_id	timestamp	latitude	longitude	altitude	provider
1	13090636733242	-35.64771771	84.51398821	131	gps
1	13090636749481	-35.64738811	84.51711292	135	gps
2	13090732131909	12.58181929	30.38181219	248	gps
2	13090732148219	12.55118199	30.38171315	246	gps

Basic RunDatabaseHelper Class

```
public class RunDatabaseHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "runs.sqlite";
   private static final int VERSION = 1;
   private static final String TABLE_RUN = "run";
   private static final String COLUMN_RUN_START_DATE = "start_date";
   public RunDatabaseHelper(Context context) {
      super(context, DB_NAME, null, VERSION);
   }
   @Override
   public void onCreate(SQLiteDatabase db) {
      // Create the "run" table
      db.execSQL("create table run (" +
         "_id integer primary key autoincrement, start_date integer)");
      // Create the "location" table
      db.execSQL("create table location (" +
         " timestamp integer, latitude real, longitude real, altitude real," +
         " provider varchar(100), run_id integer references run(_id))");
   }
   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      // Implement schema changes and data massage here when upgrading
   }
   public long insertRun(Run run) {
      ContentValues cv = new ContentValues();
      cv.put(COLUMN_RUN_START_DATE,
         run.getStartDate().getTime());
      return getWritableDatabase().insert(TABLE_RUN, null, cv);
   }
}
```

In **onCreate(…)** your job is to establish the schema for a newly created database.

In **onUpgrade(…)** you have the opportunity to execute migration code to move from one version of the database schema to another.

Runs have a single data field, their start date, and here you are storing the long value of it in the database using a **ContentValues** object to represent the mapping of column names to values.

Adding an ID to Run

```
public class Run {
    private long mId;
    private Date mStartDate;

    public Run() {
        mId = -1;
        mStartDate = new Date();
    }

    public long getId() {
        return mId;
    }

    public void setId(long id) {
        mId = id;
    }

    public Date getStartDate() {
        return mStartDate;
    }
```

To support querying one or more runs from the database, and distinguishing them in the app, you will need to add an ID property to the **Run** class.

Do some enhancements to RunManager that make use of the new database.

This will be the API that the rest of the application uses to store and retrieve data.

To start, add just enough code to get Run storage working. (See next slide)

Managing the Current Run 1/2

(RunManager.java)

```
public class RunManager {
    private static final String TAG = "RunManager";

    private static final String PREFS_FILE = "runs";
     private static final String PREF_CURRENT_RUN_ID =
            "RunManager.currentRunId";
    public static final String ACTION_LOCATION =
        "com.bignerdranch.android.runtracker.ACTION_LOCATION";

    private static final String TEST_PROVIDER = "TEST_PROVIDER";

    private static RunManager sRunManager;
    private Context mAppContext;
    private LocationManager mLocationManager;
    private RunDatabaseHelper mHelper;
     private SharedPreferences mPrefs;
     private long mCurrentRunId;
```

```
private RunManager(Context appContext) {
        mAppContext = appContext;
        mLocationManager = (LocationManager)mAppContext
            .getSystemService(Context.LOCATION_SERVICE);
        mHelper = new RunDatabaseHelper(mAppContext);
         mPrefs = mAppContext.
             getSharedPreferences(PREFS_FILE, Context.MODE_PRIVATE);
         mCurrentRunId = mPrefs.getLong(PREF_CURRENT_RUN_ID, -1);
    }
    ...

private void broadcastLocation(Location location) {
        Intent broadcast = new Intent(ACTION_LOCATION);
        broadcast.putExtra(LocationManager.KEY_LOCATION_CHANGED,
            location);
        mAppContext.sendBroadcast(broadcast);
}
```

Managing the Current Run 2/2

```
public Run startNewRun() {
        Run run = insertRun();
        startTrackingRun(run);
        return run;
    }
    public void startTrackingRun(Run run) {
        mCurrentRunId = run.getId();
        mPrefs.edit().putLong(PREF_CURRENT_RUN_ID,
                mCurrentRunId).commit();
        startLocationUpdates();
    }
    public void stopRun() {
        stopLocationUpdates();
        mCurrentRunId = -1;
        mPrefs.edit().remove(PREF_CURRENT_RUN_ID).commit();
    }
    private Run insertRun() {
        Run run = new Run();
        run.setId(mHelper.insertRun(run));
        return run;
    }
}
```

The startNewRun() method calls **insertRun()** to create and insert a new Run into the database, passes it to **startTrackingRun(Run)** to begin tracking it, and finally returns it to the caller.

You will use this method in **RunFragment** in response to the Start button when there is no existing run to work with.

RunFragment will also use **startTrackingRun(Run)** directly when it restarts tracking on an existing run.

This method saves the ID of the **Run** passed to it in both an instance variable and in shared preferences.

Storing it this way allows it to be retrieved later, even if the app is killed completely; the **RunManager** constructor will do this work in that event.

Updating the Starting and Stopping Code

(RunFragment.java)

```
@Override
  public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_run, container, false);
    ...
    mStartButton = (Button)view.findViewById(R.id.run_startButton);
    mStartButton.setOnClickListener(new View.OnClickListener() {
       @Override
       public void onClick(View v) {
          ~~mRunManager.startLocationUpdates();~~
          ~~mRun = new Run();~~
          mRun = mRunManager.startNewRun();
          updateUI();
       }
    });
    mStopButton = (Button)view.findViewById(R.id.run_stopButton);
    mStopButton.setOnClickListener(new View.OnClickListener() {
       @Override
       public void onClick(View v) {
          ~~mRunManager.stopLocationUpdates();~~
          mRunManager.stopRun();
          updateUI();  }});
    updateUI();
    return view;
  }
```

Making use of the new **RunManager** methods in **RunFragment**.

Inserting Locations in the Database

(RunDatabaseHelper.java)

```
public class RunDatabaseHelper extends SQLiteOpenHelper {
   ...
   private static final String TABLE_RUN = "run";
   private static final String COLUMN_RUN_START_DATE = "start_date";
   private static final String TABLE_LOCATION = "location";
   private static final String COLUMN_LOCATION_LATITUDE = "latitude";
   private static final String COLUMN_LOCATION_LONGITUDE = "longitude";
   private static final String COLUMN_LOCATION_ALTITUDE = "altitude";
   private static final String COLUMN_LOCATION_TIMESTAMP = "timestamp";
   private static final String COLUMN_LOCATION_PROVIDER = "provider";
   private static final String COLUMN_LOCATION_RUN_ID = "run_id";

public long insertLocation(long runId, Location location) {
      ContentValues cv = new ContentValues();
      cv.put(COLUMN_LOCATION_LATITUDE, location.getLatitude());
      cv.put(COLUMN_LOCATION_LONGITUDE, location.getLongitude());
      cv.put(COLUMN_LOCATION_ALTITUDE, location.getAltitude());
      cv.put(COLUMN_LOCATION_TIMESTAMP, location.getTime());
      cv.put(COLUMN_LOCATION_PROVIDER, location.getProvider());
      cv.put(COLUMN_LOCATION_RUN_ID, runId);
      return getWritableDatabase().insert(TABLE_LOCATION, null, cv);
 } ...
```

Similar to inserting **Runs**, you will add a method to both **RunDatabaseHelper** and **RunManager** to insert a location for the current run.

Unlike inserting a **Run**, however, **RunTracker** needs to be able to insert locations as updates arrive regardless of whether the user interface is visible or the application is running.

To handle this requirement, a standalone **BroadcastReceiver** is the best option.

Inserting a Location for the Current Run

Add code to RunManager to insert a location for the currently tracking run.
(RunManager.java)

```
 private Run insertRun() {
        Run run = new Run();
        run.setId(mHelper.insertRun(run));
        return run;
    }

    public void insertLocation(Location loc) {
        if (mCurrentRunId != -1) {
            mHelper.insertLocation(mCurrentRunId, loc);
        } else {
            Log.e(TAG, "Location received with no tracking run; ignoring.");
        }
    }
}
```

Lastly, **insertLocation(Location)** method in a standalone **BroadcastReceiver**. You can guarantee that your location intents will be handled no matter whether the rest of **RunTracker** is up and running.

(TrackingLocationReceiver.java)

```
public class TrackingLocationReceiver
        extends LocationReceiver {
   protected void onLocationReceived(Context c,
           Location loc) {
       RunManager.get(c).insertLocation(loc);
   }
}
```

(AndroidManifest.xml)

```
<application
  ...>
  ...
~~<receiver android:name=".LocationReceiver"~~
  <receiver android:name=".TrackingLocationReceiver"
    android:exported="false">
    <intent-filter>
      <action android:name=
     "com.bignerdranch.android.runtracker.ACTION_LOCATION"/>
    </intent-filter>
  </receiver>
</application>
```

Querying a List of Runs From the Database 1/3

Querying a **SQLiteDatabase** returns an instance of **Cursor** describing the results. Cursors treat their results as a series of rows and columns, and only support Strings and primitive types for values.

Since you already have database tables that represent your objects, it would be ideal if you could get instances of those objects from a **Cursor**.

CursorWrapper is designed to wrap an existing **Cursor** and forward along all of the method calls to it. On its own, it is not very useful; but as a superclass, it can provide you with a good foundation on which to build your own custom cursor implementations specific for your model objects.

Update **RunDatabaseHelper** to include a new **queryRuns()** method that returns a **RunCursor** listing all the runs in order by date - an example of the **CursorWrapper** pattern.

Querying a List of Runs From the Database 2/3

```
public class RunDatabaseHelper extends SQLiteOpenHelper {
   private static final String DB_NAME = "runs.sqlite";
   private static final int VERSION = 1;
   private static final String TABLE_RUN = "run";
   private static final String COLUMN_RUN_ID = "_id";
   private static final String COLUMN_RUN_START_DATE = "start_date";
   ...
   public RunCursor queryRuns() {
      // Equivalent to "select * from run order by start_date asc"
      Cursor wrapped = getReadableDatabase().query(TABLE_RUN,
            null, null, null, null, null, COLUMN_RUN_START_DATE + " asc");
      return new RunCursor(wrapped);
   }
   /**
    * A convenience class to wrap a cursor that returns rows from the "run" table.
    * The {@link getRun()} method will give you a Run instance representing
    * the current row.
    */
   public static class RunCursor extends CursorWrapper {
      public RunCursor(Cursor c) {
         super(c);
      }
      /**
       * Returns a Run object configured for the current row,
       * or null if the current row is invalid.
       */
      public Run getRun() {
         if (isBeforeFirst() || isAfterLast()) return null;
         Run run = new Run();
         long runId =
               getLong(getColumnIndex(COLUMN_RUN_ID));
         run.setId(runId);
         long startDate = getLong(getColumnIndex(
                COLUMN_RUN_START_DATE));
         run.setStartDate(new Date(startDate));
         return run;
      }
   }
}
```

The **getRun()** method checks to ensure that the cursor is within its bounds and then creates and configures an instance of **Run** based on the values of the current row's columns.

Querying a List of Runs From the Database 3/3

A user of **RunCursor** would iterate over the rows in the result set and call **getRun()** for each row to get a nice object instead of a bunch of nasty primitives.

Thus, **RunCursor's** main purpose is to encapsulate the grunt work of turning a row in the run table into an instance of **Run**, marshaling and massaging the data as needed.

The **queryRuns()** method does the work of executing the SQL query and providing the plain cursor to a new **RunCursor**, which it returns to the caller.

Now you can make use of this new method in **RunManager** and eventually **RunListFragment**.

Proxying run queries (RunManager.java)

```
private Run insertRun() {
        Run run = new Run();
        run.setId(mHelper.insertRun(run));
        return run;
    }
    public RunCursor queryRuns() {
        return mHelper.queryRuns();
    }
    public void insertLocation(Location loc) {
        if (mCurrentRunId != -1) {
            mHelper.insertLocation(mCurrentRunId, loc);
        } else {
            Log.e(TAG,
                  “Location received with no tracking run; ignoring.");
        }
    }
```

Displaying a List of Runs Using CursorAdapter

To set the ground work for a user interface listing the runs, create a new default activity for the app called **RunListActivity**.

```
public class RunListActivity  extends SingleFragmentActivity {
   protected Fragment createFragment() {
      return new RunListFragment();
   }
}
```

Configuring RunListActivity (AndroidManifest.xml)

```
<application ...>
 ~~<activity android:name=".RunActivity"~~
 <activity android:name=".RunListActivity"
   android:label="@string/app_name">
   <intent-filter>
     <action android:name="android.intent.action.MAIN" />
     <category android:name="android.intent.category.LAUNCHER" />
   </intent-filter></activity>
 <activity android:name=".RunActivity"
         android:label="@string/app_name" />
 <receiver android:name=".TrackingLocationReceiver"
   android:exported="false">  ...
```

Basic Skeleton of RunListFragment

```
public class RunListFragment extends ListFragment {
   private RunCursor mCursor;
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      // Query the list of runs
      mCursor = RunManager.get(getActivity()).queryRuns();
   }
   @Override
   public void onDestroy() {
      mCursor.close();
      super.onDestroy();
   }
}
```

You are loading the cursor in **onCreate(Bundle)** and closing it in **onDestroy().**

Implementing RunCursorAdapter 1/2

```
public class RunListFragment extends ListFragment {
    private RunCursor mCursor;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Query the list of runs
         mCursor = RunManager.get(getActivity()).queryRuns();
         // Create an adapter to point at this cursor
         RunCursorAdapter adapter =
                 new RunCursorAdapter(getActivity(), mCursor);
         setListAdapter(adapter);
    }
    @Override
    public void onDestroy() {
        mCursor.close();
        super.onDestroy();
    }
    private static class RunCursorAdapter extends CursorAdapter {
        private RunCursor mRunCursor;
        public RunCursorAdapter(Context context, RunCursor cursor) {
            super(context, cursor, 0);
            mRunCursor = cursor;
        }
        @Override
        public View newView(Context context, Cursor cursor,
             ViewGroup parent) {
            // Use a layout inflater to get a row view
            LayoutInflater inflater = (LayoutInflater)context
                .getSystemService(
                        Context.LAYOUT_INFLATER_SERVICE);
            return inflater
                .inflate(android.R.layout.simple_list_item_1, parent, false);
        }
        @Override
        public void bindView(View view, Context context,
             Cursor cursor) {
            // Get the run for the current row
            Run run = mRunCursor.getRun();
            // Set up the start date text view
            TextView startDateTextView = (TextView)view;
            String cellText =
                context.getString(R.string.cell_text, run.getStartDate());
            startDateTextView.setText(cellText);
        }
    }
}
```

Implementing RunCursorAdapter 2/2

The constructor for **CursorAdapter** takes a **Context**, a **Cursor**, and an integer of flags. Most of the flags are now deprecated or questionable in favor of using loaders, so here you pass zero. You also stash the **RunCursor** in an instance variable to avoid having to cast it later.

Next, you implement **newView(Context, Cursor, ViewGroup)** to return a View to represent the current row in the cursor. This example inflates the system resource android.R.layout.simple_list_item_1, which is a plain **TextView**. Since all of the views in the list will look the same, that is all the logic you need here.

The **bindView(View, Context, Cursor)** method will be called by **CursorAdapter** when it wants you to configure a view to hold data for a row in the cursor. This will always be called with a View that has been previously returned from **newView(…).**

Implementing **bindView(…)** is relatively simple. First, you ask the **RunCursor** for the Run at the current row (the cursor will have already been positioned by CursorAdapter). Next, you assume that the view passed in is a **TextView** and configure it to display a simple description of the Run.

With these changes in place, run RunTracker; you should see a list of the runs you have previously created, assuming you ran it and started a run.

Creating New Runs 1/2

Options menu for the run list (run_list_options.xml)

```
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android" >
  <item android:id="@+id/menu_item_new_run"
    android:showAsAction="always"
    android:icon="@android:drawable/ic_menu_add"
    android:title="@string/new_run"/>
</menu>
```

Adding a New Run string (strings.xml)

```
<string name="stop">Stop</string>
  <string name="gps_enabled">GPS Enabled</string>
  <string name="gps_disabled">GPS Disabled</string>
  <string name="cell_text">Run at %1$s</string>
    <string name="new_run">New Run</string>
</resources>
```

```
public class RunListFragment extends ListFragment {
  private static final int REQUEST_NEW_RUN = 0;

  private RunCursor mCursor;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setHasOptionsMenu(true);
    // Query the list of runs
    mCursor = RunManager.get(getActivity()).queryRuns();
    // Create an adapter to point at this cursor
    RunCursorAdapter adapter =
        new RunCursorAdapter(getActivity(), mCursor);
    setListAdapter(adapter);
  }

  ...
```

Creating New Runs 2/2

```
@Override
    public void onCreateOptionsMenu(Menu menu,
                    MenuInflater inflater) {
        super.onCreateOptionsMenu(menu, inflater);
        inflater.inflate(R.menu.run_list_options, menu);
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.menu_item_new_run:
            Intent i = new Intent(getActivity(), RunActivity.class);
            startActivityForResult(i, REQUEST_NEW_RUN);
            return true;
        default:
            return super.onOptionsItemSelected(item);
        }
    }
```

```
@Override
public void onActivityResult(int requestCode,
            int resultCode, Intent data) {
    if (REQUEST_NEW_RUN == requestCode) {
        mCursor.requery();
        ((RunCursorAdapter)getListAdapter()).notifyDataSetChanged();
    }
 }
private static class RunCursorAdapter extends CursorAdapter {

    private RunCursor mRunCursor;
```

The only thing new and interesting about this approach is that you use **onActivityResult(…)** to force the list to reload once the user returns to it after navigating elsewhere.

Working with Existing Runs

The next logical thing to do is allow the user to navigate from the list of runs to the details of a specific run. **RunFragment** needs some support for passing a run ID as an argument. Since you host **RunFragment** in **RunActivity**, it too needs an extra for the run ID.

```
public class RunFragment extends Fragment {
    private static final String TAG = "RunFragment";
    private static final String ARG_RUN_ID = "RUN_ID";
    ...
    private TextView mStartedTextView, mLatitudeTextView,
        mLongitudeTextView, mAltitudeTextView, mDurationTextView;

    public static RunFragment newInstance(long runId) {
        Bundle args = new Bundle();
        args.putLong(ARG_RUN_ID, runId);
        RunFragment rf = new RunFragment();
        rf.setArguments(args);
        return rf;
    }
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

Adding a Run ID Extra

(RunActivity.java)

```
public class RunActivity extends SingleFragmentActivity {
   /** A key for passing a run ID as a long */
   public static final String EXTRA_RUN_ID =
      "com.bignerdranch.android.runtracker.run_id";
   @Override
   protected Fragment createFragment() {
      ~~return new RunFragment();~~
      long runId = getIntent().getLongExtra(EXTRA_RUN_ID, -1);
      if (runId != -1) {
         return RunFragment.newInstance(runId);
      } else {
         return new RunFragment();
      }
   }
}
```

Now, make use of the new fragment convention in **RunActivity**. If the intent has a RUN_ID extra, create the **RunFragment** using **newInstance(long)**. If not, just use the default constructor as before.

Launching existing runs via onListItemClick(…)

```
@Override
   public void onListItemClick(ListView l, View v,
         int position, long id) {
      // The id argument will be the Run ID; CursorAdapter
      //   gives us this for free
      Intent i = new Intent(getActivity(), RunActivity.class);
      i.putExtra(RunActivity.EXTRA_RUN_ID, id);
      startActivity(i);
   }

   private static class RunCursorAdapter extends CursorAdapter
{
```

There is a small bit of magic in play here. Because you named the ID column in the run table _id, CursorAdapter has detected it and passed it as the id argument to **onListItemClick(…).**

Querying a Single Run

Starting RunFragment with an ID argument is not enough to get it to display details about an existing run.

You need to query the database for the details of the existing run, including its last recorded location, in order to populate the user interface.
(RunDatabaseHelper.java)

```
public RunCursor queryRun(long id) {
        Cursor wrapped = getReadableDatabase()
                .query(TABLE_RUN,
            null, // All columns
            COLUMN_RUN_ID + " = ?", // Look for a run ID
            new String[]{ String.valueOf(id) }, // with this value
            null, // group by
            null, // order by
            null, // having
            "1"); // limit 1 row
        return new RunCursor(wrapped);
}
```

Next, you will add a **getRun(long)** method to **RunManager** that wraps the results of the **queryRun(long)** method and pulls a Run out of the first row, if it has one.

Implementing getRun(long) (RunManager.java)
Launching existing runs via onListItemClick(...)

```
public Run getRun(long id) {
        Run run = null;
        RunCursor cursor = mHelper.queryRun(id);
        cursor.moveToFirst();
        // If you got a row, get a run
        if (!cursor.isAfterLast())
            run = cursor.getRun();
        cursor.close();
        return run;
    }
```

Update RunFragment to Work with Run 1/2

```
public class RunFragment extends Fragment {
    private static final String TAG = "RunFragment";
    private static final String ARG_RUN_ID = "RUN_ID";

    private BroadcastReceiver mLocationReceiver =
        new LocationReceiver() {
        @Override
        protected void onLocationReceived(Context context,
             Location loc) {
            if (!mRunManager.isTrackingRun(mRun))
                return;
            mLastLocation = loc;
            if (isVisible())
                updateUI();
        }
        @Override
        protected void onProviderEnabledChanged(
               boolean enabled) {
            int toastText = enabled ? R.string.gps_enabled :
                                              R.string.gps_disabled;
            Toast.makeText(getActivity(), toastText,
                 Toast.LENGTH_LONG).show();
        }
    };
    ...
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        mRunManager = RunManager.get(getActivity());
       // Check for a Run ID as an argument, and find the run
         Bundle args = getArguments();
         if (args != null) {
             long runId = args.getLong(ARG_RUN_ID, -1);
             if (runId != -1) {
                 mRun = mRunManager.getRun(runId);
             }
         }
    }
```

Update RunFragment to Work with Run 2/2

```
@Override
public View onCreateView(LayoutInflater inflater,
     ViewGroup container, Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_run, container, false);
   ...
   mStartButton = (Button)view.findViewById(R.id.run_startButton);
   mStartButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
         ~~mRun = mRunManager.startNewRun();~~
         if (mRun == null) {
            mRun = mRunManager.startNewRun();
         } else {
            mRunManager.startTrackingRun(mRun);
         }
         updateUI();
      }
   });
   ...
   return view;
}
...
```

```
private void updateUI() {
     boolean started = mRunManager.isTrackingRun();
     boolean trackingThisRun = mRunManager.isTrackingRun(mRun);
     if (mRun != null)
        mStartedTextView.setText(mRun.getStartDate().toString());
     int durationSeconds = 0;
     if (mRun != null && mLastLocation != null) {
        durationSeconds = mRun.getDurationSeconds(
              mLastLocation.getTime());
        mLatitudeTextView.setText(Double.toString(
              mLastLocation.getLatitude()));
        mLongitudeTextView.setText(Double.toString(
               mLastLocation.getLongitude()));
        mAltitudeTextView.setText(Double.toString(
               mLastLocation.getAltitude()));
     }
     mDurationTextView.setText(Run.formatDuration(durationSeconds));
     mStartButton.setEnabled(!started);
     ~~mStopButton.setEnabled(started);~~
     mStopButton.setEnabled(started && trackingThisRun);
   }
}
```

Update RunFragment to Work with Run 2/2

```
@Override
public View onCreateView(LayoutInflater inflater,
      ViewGroup container, Bundle savedInstanceState) {
   View view = inflater.inflate(R.layout.fragment_run, container, false);
   ...
   mStartButton = (Button)view.findViewById(R.id.run_startButton);
   mStartButton.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
         ~~mRun = mRunManager.startNewRun();~~
         if (mRun == null) {
            mRun = mRunManager.startNewRun();
         } else {
            mRunManager.startTrackingRun(mRun);
         }
         updateUI();
      }
   });
   ...
   return view;
}
...
```

```
private void updateUI() {
      boolean started = mRunManager.isTrackingRun();
      boolean trackingThisRun = mRunManager.isTrackingRun(mRun);
      if (mRun != null)
         mStartedTextView.setText(mRun.getStartDate().toString());
      int durationSeconds = 0;
      if (mRun != null && mLastLocation != null) {
         durationSeconds = mRun.getDurationSeconds(
               mLastLocation.getTime());
         mLatitudeTextView.setText(Double.toString(
               mLastLocation.getLatitude()));
         mLongitudeTextView.setText(Double.toString(
               mLastLocation.getLongitude()));
         mAltitudeTextView.setText(Double.toString(
               mLastLocation.getAltitude()));
      }
      mDurationTextView.setText(Run.formatDuration(durationSeconds));
      mStartButton.setEnabled(!started);
      ~~mStopButton.setEnabled(started);~~
      mStopButton.setEnabled(started && trackingThisRun);
   }
}
```

Querying the Last Location for a Run 1/2

You can make **RunFragment** load the last location for the current run from the database.

This will be very similar to loading the **Run**, but you will create a new **LocationCursor** to work with Location objects along the way.

```
public LocationCursor queryLastLocationForRun(long runId)   {
    Cursor wrapped =
        getReadableDatabase().query(TABLE_LOCATION,
         null, // All columns
         COLUMN_LOCATION_RUN_ID + " = ?", // limit to given run
         new String[]{ String.valueOf(runId) },
         null, // group by
         null, // having
         COLUMN_LOCATION_TIMESTAMP + " desc", // Latest first
         "1"); // limit 1
    return new LocationCursor(wrapped);
  }
```

```
public static class LocationCursor extends CursorWrapper {
    public LocationCursor(Cursor c) {  super(c);   }
    public Location getLocation() {
      if (isBeforeFirst() || isAfterLast()) return null;
      // First get the provider out so you can use the constructor
      String provider = getString(getColumnIndex(
          COLUMN_LOCATION_PROVIDER));
      Location loc = new Location(provider);
      loc.setLongitude(getDouble(getColumnIndex(
          COLUMN_LOCATION_LONGITUDE)));
      loc.setLatitude(getDouble(getColumnIndex(
          COLUMN_LOCATION_LATITUDE)));
      loc.setAltitude(getDouble(getColumnIndex(
          COLUMN_LOCATION_ALTITUDE)));
      loc.setTime(getLong(getColumnIndex(
          COLUMN_LOCATION_TIMESTAMP)));
      return loc;
    }
  }
```

Note that **Location's** constructor requires the provider name, so you first pull that out of the current row before setting up the rest of the properties.

Querying the Last Location for a Run 2/2

(RunManager.java)

```
 public Location getLastLocationForRun(long runId) {
        Location location = null;
        LocationCursor cursor = mHelper
              .queryLastLocationForRun(runId);
        cursor.moveToFirst();
        // If you got a row, get a location
        if (!cursor.isAfterLast())
            location = cursor.getLocation();
        cursor.close();
        return location;
    }
```

Just as with **queryRun(long)**, you should create a method in **RunManager** to wrap it and return a **Location** from the one row in the cursor.

Now you can use this new method in **RunFragment** to fetch the last location for the current run when the fragment is created.

(RunFragment.java)

```
@Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true);
        mRunManager = RunManager.get(getActivity());

        // Check for a Run ID as an argument, and find the run
        Bundle args = getArguments();
        if (args != null) {
            long runId = args.getLong(ARG_RUN_ID, -1);
            if (runId != -1) {
                mRun = mRunManager.getRun(runId);
                mLastLocation = mRunManager.getLastLocationForRun(runId);
            }
        }
    }
```

Now, after much ado, you should have a **RunTracker** that is capable of creating and tracking as many runs as your device’s disk (and battery) can withstand and displaying them to the user in a logical fashion.

Challenge: Identifying the Current Run

As implemented, the only way to identify which run is being tracked is to manually visit it from the list and see that the start and stop buttons are enabled appropriately. It would be great if the user had an easier way to access the current run.

For a simple challenge, give the list row for the current run a different treatment in the UI, like an icon or color change.

For a harder challenge, use an ongoing notification to let the user know that you are tracking them, and launch the **RunActivity** when it is pressed.