ECE155: Engineering Design with Embedded Systems Winter 2016

Lecture 7 — Android II

Patrick Lam

Programming Tips

First, a programming tip for object-oriented systems. Sometimes, you have code in a class, say an `EventListener`, which needs access to the object that created it, say the `MainActivity`. Add a field to the `EventListener` with a reference to the `MainActivity`, as follows:

```
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    // ...
    LightSensorEventListener el = new LightSensorEventListener(!{\bf this}!);
  }
}

class LightSensorEventListener implements SensorEventListener {
  MainActivity m;

  public LightSensorEventListener(MainActivity m) {
      this.m = m;
  }

  // ...
}
```

A second tip: don't put your app in the `com.example` namespace. Put it in `ca.uwaterloo`.

Intents

So far, we've seen Activities in isolation, and that'll be all you need for the labs. However, real apps often have more than one Activity. Today's topic is going to be about how to link Activities. Android uses Intents.

An *Intent* specifies a request or describes an event.

Examples: Starting another Activity. The simplest possible Intent explicitly names the Activity it would like to start.

```
Intent intent = new Intent(this, OtherActivity.class);
startActivity(intent);
```

Examples: Requests. The Map activity puts up hyperlinks to webpages and phone numbers. Upon click, the Map broadcasts either the web browser Intent or the call Intent and trusts that some other application will handle the Intent.

Examples: Events. The system broadcasts an Intent when the phone goes into airplane mode.

How Intents Work. One component broadcasts an Intent. Then, 0 or more components receive the Intent. Android may pick a component to act upon the Intent.

Parts of an Intent. Each intent contains an *action* which represents the requested action, along with optional data. For instance:

ACTION_MAIN	Launch an activity
ACTION_DIAL	Dial a phone number
ACTION_SEARCH	Perform a search

The two following code fragments yield identical Intents with an action (but no data):

```
new Intent(Intent.ACTION_EDIT);   |  Intent intent = new Intent();
                                  |  intent.setAction(Intent.ACTION_EDIT);
```

The *data* is the payload of the event. Android intent data is in URI (Universal Resource Identifier) format[1]. The obvious thing to put into data is a web address, but other data formats are possible as well:

```
new Intent(Intent.ACTION_DIAL,           | Intent intent =
                                         |   new Intent(Intent.ACTION_DIAL);
           Uri.parse("tel:6175551212")); | intent.setData(
                                         |   Uri.parse("tel:6175551212"));
```

You can also tell Android what type of data to expect it to find at the URI, using the `setType` method. This is optional. Example: `intent.setType("audio/mp3");`

Beyond the data, Intents may also contain *extras*. The extras consist of key-value pairs: they contain more information than what you can easily put into a URI. Here's an example.

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(android.content.Intent.EXTRA_EMAIL,
  new String[] {
    "p.lam@ece.uwaterloo.ca", "root@uwaterloo.ca"
  });
```

(Key-value pairs are a key concept you'll encounter over and over, by the way.)

Finally, Intents may also contain *flags*, which modify how the Intent gets launched and how it will be processed by the recipient. They don't affect which activity gets launched. Examples: `FLAG_ACTIVITY_NO_HISTORY` and `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED`.

Intent Resolution

We've seen how to explicitly name which Activity we want to launch, and I've alluded to how we can implicitly launch an activity by describing what we're looking for. In either case, we use the `startActivity` method to launch the intent.

In the case of implicit intent resolution, the system searches the available Activities on a system, using the Intent's action, data and category.

[1] What's a URI? It's like a URL. The exact distinction is unimportant for ECE155.

Requesting Intent Resolution. Let's first look at an example of requesting an image from any suitable Activity on the system [Vog13]:

```
private static final int REQUEST_CODE = 1;

public void pickImage(View View) {
  Intent intent = new Intent();
  intent.setType("image/*");
  intent.setAction(Intent.ACTION_GET_CONTENT);
  intent.addCategory(Intent.CATEGORY_OPENABLE);
  startActivityForResult(intent, REQUEST_CODE);
}
```

This specifies a requested type, action, and category, and asks that some other activity returns an image. We'll see how to process activity results below.

Responding to Intent Resolution Requests. In your application's manifest, there's an XML tag for each activity. That tag can take an *intent filter* describing the Intents that the activity is prepared to handle. For example:

```
<activity
    android:name=".BrowserActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />

        <category android:name="android.intent.category.DEFAULT" />

        <data android:scheme="http" />
    </intent-filter>
</activity>
```

This activity is able to view any URI that begins with `http`.

Receiving Activity results. When you launch a sub-Activity, sometimes you're hoping for a result back from that activity. For instance, you may be asking for the user to pick a contact from the contact list. Or you may be asking the user for a favourite colour. The sub-activity should create a new Intent and call its `setResult` method:

```
Intent retval = new Intent();
retval.putExtra("color", "blue");
setResult(RESULT_OK, retval);
```

Then in the caller, we provide a new callback, `onActivityResult()`.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
  if (resultCode == RESULT_OK && requestCode == YOUR_REQUEST_CODE) {
    if (data.hasExtra("color")) {
      String favouriteColor = data.getExtras().getString("color");
    }
```

```
   }
  }
```

The `resultCode` is an `int` you provided while starting the sub-activity. It allows you to distinguish different sub-activities you may have started.

For a video tutorial on Intents, see: `https://www.youtube.com/watch?v=mizDa8bkbZ4`. This followup video gives an example of passing data via Intents: `https://www.youtube.com/watch?v=0ism1OM0an0`. Both are worth watching.

Saving and Restoring State

Sometimes, Android kills your activity but brings it back later. You want the activity to have the same data when it comes back from the dead. This will happen by default for anything in a UI element, but not for any fields stored in the activity (like, say, step counts). In your activity, implement the callback `onSaveInstanceState()` to save the data:

```
class MainActivity ... {
  String color;

  @Override
  protected void onSaveInstanceState(Bundle b) {
    super.onSaveInstanceState(b);
    b.putString("color", color);
  }
}
```

The Bundle `b` is another key/value map. You can then restore whatever data you saved in the `onCreate` method of the same Activity:

```
class MainActivity ... {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    if (savedInstanceState != null)
      color = b.getString("color");
  }
}
```

More Android Concepts

Two additional concepts: when to use XML versus Java code for adding widgets to your layouts; and what "inflate" means in the code that you always auto-generate.

XML versus Java. Use the right tool for the job! Sometimes this is XML; other times, it's Java. Here's the tradeoff.

XML = more safety:

- You can select and place items on your layouts at any point.

- You don't need to run your code under the emulator (or on a phone) to see how things will look.
- You get more error checking: the compiler can tell you about what you're doing wrong and help you fix it ahead of time.

Java Code = more flexibility:

- You can choose widgets based on user input or your computations.
- You can use loops or other control flow to generate related items.
- You get less error checking of what you've written.

What "inflate" means. These lines keep on showing up in our auto-generated `Activity` code:

```
// Inflate the menu; this adds items to the action bar if it is present.
getMenuInflater().inflate(R.menu.activity_main, menu);
```

"Inflate" means taking an XML tree and creating a tree of `View` objects, based on the description in the XML.

Android Graphics

Next, we'll see how to put graphics on the screen in Android. There are, in general, two options:

- use a `View` (easier; but suitable only for infrequent updates); or
- paint to a `Canvas` (more complicated; but permits real-time updates).

Even though the `Canvas` is more suitable for embedded systems, the `View` is easier, so we'll be talking about it in this class.

Drawing to a `View`.

As always with Android, you can put things onto the `View` either by specifying XML, or programmatically. In either case, you will use a `Drawable`, either explicitly or implicitly.

The `Drawable` class. A `Drawable` object represents, as its name suggests, "something that can be drawn". Examples include:

- BitmapDrawable: draw a bitmap onto the screen;
- ShapeDrawable: draw a shape;
- PictureDrawable: play back a sequence of drawing calls.
- etc.

All of the drawing techniques we're going to see are going to create a `Drawable` of some sort.

Drawing Bitmaps. The easiest way to get some graphics onto your screen is by drawing them in some other program and then including them:

- put a picture (`PNG`, `JPG` or `GIF`) in `res/drawables`; and
- use an `ImageView` to include it on the screen.

Examples from [Gui13]

```
// Instantiate an ImageView and define its properties programmatically
ImageView i = new ImageView(this);
i.setImageResource(R.drawable.my_image);
 // set the ImageView bounds to match the Drawable's dimensions
i.setAdjustViewBounds(true);
i.setLayoutParams(new Gallery.LayoutParams
    (LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT));

// Add the ImageView to the layout and
//  set the layout as the content view
mLinearLayout.addView(i);
```

Or, you can do it through XML. Again, you need the appropriate drawable in the `res/drawables` directory. Note that since we're using XML, it's harder to go wrong.

```
<ImageView
  android:id="@+id/imageView1"
  android:layout_height="wrap_content"
  android:layout_width="wrap_content"
  android:src="@drawable/myImage" />
```

`ShapeDrawable`. Next, we'll see how to draw our own shapes. Primitive shapes include:

- PathShape—lines;
- RectShape—rectangles;
- OvalShape—ovals and rings;

Once again, we put these into an `ImageView`.

You can draw shapes in XML and put them into an `ImageView`. Again, you need the appropriate drawable in the `res/drawables` directory. Add this snippet, which creates the `ImageView` backed by the drawable to your Layout XML:

```
<ImageView android:id="@+id/imageView2"
  android:src="@drawable/cyan_shape" ... />
```

Next, create a separate XML file for the drawable itself. This actually provides instructions for Android to put dots on the screen.

```
<shape android:shape="oval" ... >
  <size android:width="160px" android:height="160px" />
  <solid android:color="#7f00ffff" />
</shape>
```

Everything you can do in XML, you can do in code. Here, we specify the shape in Java.

```
private class MyDrawableView extends ImageView {
  private ShapeDrawable mDrawable;
  public MyDrawableView(Context context, int color) {
    ...
    mDrawable = new ShapeDrawable(new OvalShape());
    mDrawable.getPaint().setColor(color);
    mDrawable.setBounds(0, 0, size, size);
    mDrawable.setAlpha(alpha);
  }
  protected void onDraw(Canvas canvas) {
    mDrawable.draw(canvas);
  }
}
```

Then, in the Activity's `onCreate()`, we would instantiate this `MyDrawableView` and add it to the parent view:

```
MyDrawableView magentaView =
  new MyDrawableView(this, Color.MAGENTA);
magentaView.setLayoutParams
  (new LinearLayout.LayoutParams(160, 160));
addView(magentaView);
```

References

[Gui13] Android Developer Guide. Canvas and drawables. `http://developer.android.com/guide/topics/graphics/2d-graphics.html`, 2013. Online; accessed 18-January-2013.

[Vog13] Lars Vogel. Android Intents - Tutorial, 2013. Online; accessed 12-December-2013. URL: `http://www.vogella.com/articles/AndroidIntent/article.html`.