Lecture 7 — Android II

Patrick Lam
p.lam@ece.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 14, 2015

Part I

Programming Tips

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:

Programming Tips

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

class LightSensorEventListener implements SensorEventListener {
  MainActivity m;

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

  // ...
}
```

Don't put your app in the `com.example` namespace.

Put it in `ca.uwaterloo`.

Part II

Android Intents

Intents

So far we only examined an Activity in isolation.

For the labs, you won't need anything more.

In real apps, there's usually multiple activities.

We link them with Intents.

Intents

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

The simplest possible Intent explicitly names the Activity it would like to start.

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

Intent: Request Example

The Map activity puts up links to webpages & phone numbers.

Upon click, the Map broadcasts either the web browser Intent or the call Intent.

The Map trusts that some other application will handle the Intent.

Intent: Event Example

The system broadcasts an Intent when the phone enters flight mode.

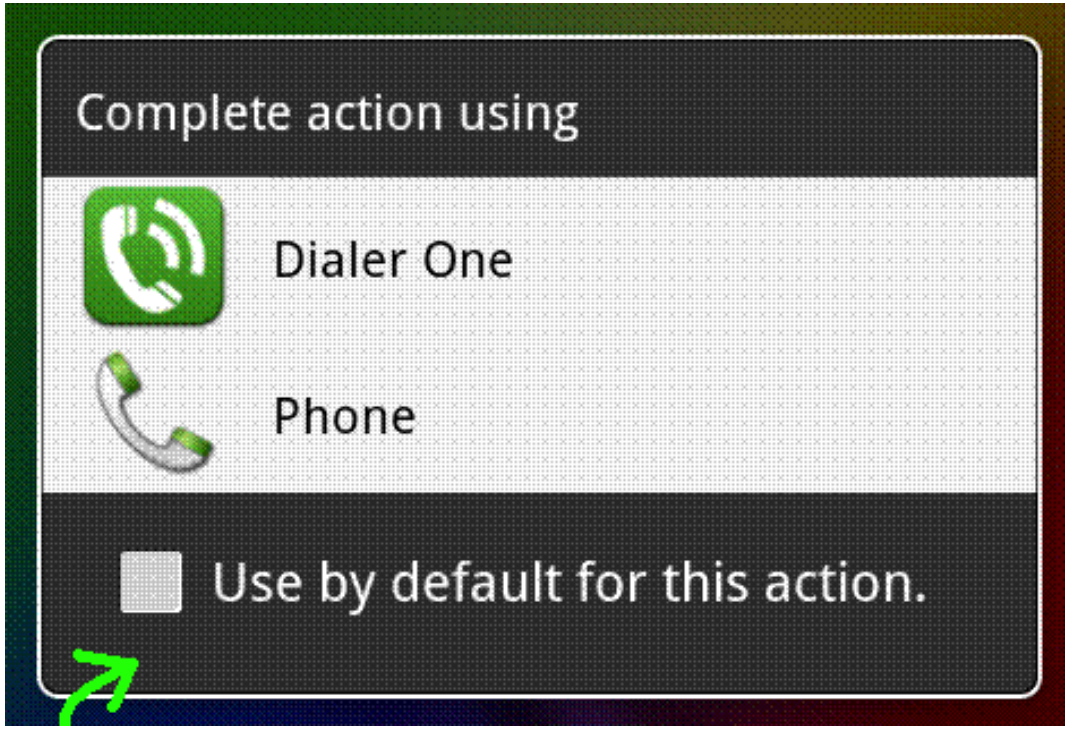
How Intents Work

One component broadcasts an Intent.

Then, 0 or more components receive the Intent.

Android may pick a component to act upon the Intent.

Choosing the Handler



(Image from stackoverflow.com)

In this case, the user is asked to choose.

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

Parts of an Intent

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);
```

Parts of an Intent

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:

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

Parts of an Intent

```
new Intent(Intent.ACTION_DIAL,
           Uri.parse("tel:6175551212"));
```

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

Parts of an Intent

You can also tell Android what type of data to expects.

Use the `setType` method. Example:

```
intent.setType("audio/mp3");
```

This is optional.

Intent Extras

Beyond the data, Intents may also contain *extras*.

Extras consist of key-value pairs.

They contain more information than what can easily be put in a URI.

Intent Extras 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"
  });
```

Intent Flags

Finally, Intents may also contain *flags*.

These 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 can explicitly name which Activity we want to launch.

We can implicitly launch an activity by describing what we want.

In either case, use `startActivity` to launch the intent.

Implicit intent resolution: system searches the available Activities, using the Intent's action, data and category.

Requesting Intent Resolution

```
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);
}
```

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);
```

Receiving Activity Results

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");
    }
 }
}
```

Receiving Activity Results

The `resultCode` is an `int` you provided while starting the sub-activity.

It allows you to distinguish different sub-activities you may have started.

Passing Data Using Intents

Video:
`https://www.youtube.com/watch?v=0ism1OM0an0`

Part III

Saving & Restoring State

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()`

Saving State

To save state:

(The Bundle b is another key/value map.)

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

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

Restoring State

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");
  }
}
```

Part IV

Graphics on Android

XML versus Programmatic Construction

Use the right tool for the job!

XML = more safety:

- Select and place items ahead of time.
- Don't need the emulator to see how things will look.
- More error checking.

Java Code = more flexibility:

- Can choose widgets based on user input or computations.
- Can use loops, etc to generate related items.
- Less error checking.

What “inflate” means

These lines keep on showing up in our code:

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

“Inflate” = taking an XML and
creating `View` objects, based on the description in the XML.

Putting Graphics on the Screen

Two choices:

- use a `View` (easier; infrequent updates); or
- paint to a `Canvas` (more complicated, many updates).

Main class: `Drawable`

Represents “something that can be drawn”, e.g.

- BitmapDrawable
- ShapeDrawable
- PictureDrawable
- etc.

Drawing to a View

As always with Android, either:

- through XML; or
- programmatically.

Bitmaps through ImageView

Easiest way[2]:

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

from: `http://developer.android.com/guide/topics/graphics/2d-graphics.html`

```
// 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);
```

[2] Thanks to `http://www.cs.umd.edu/class/fall2010/CMSC498G/CMSC498G/Slides_files/Graphics.pptx`

Bitmaps through ImageView XML

Again, you need the appropriate drawable in the `res/drawables` directory.

Note: harder to go wrong here.

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

Drawing on a ShapeDrawable

Primitive shapes:

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

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

Shapes from XML

Again, you need the appropriate drawable in the `res/drawables` directory.

Note: harder to go wrong here.

In the Layout XML:

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

Next, we create an XML for the drawable itself:

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

Shapes, programmatically

Everything you can do in XML, you can do in code.

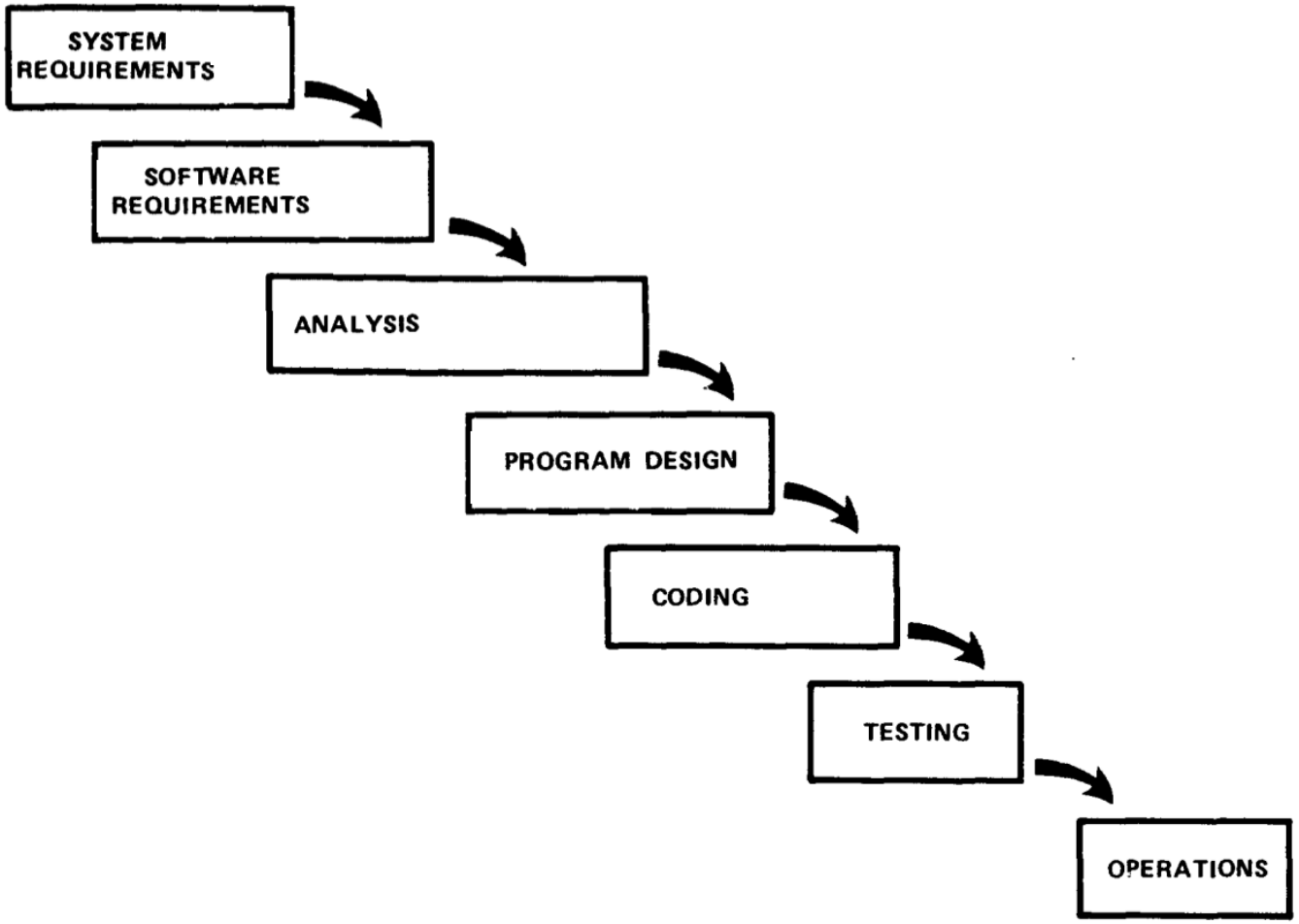
```
private class MyDrawableView extends ImageView {
  private ShapeDrawable mDrawable;
  public MyDrawableView(Context context, int color) {
    ...
    mDrawable = new ShapeDrawable(new OvalShape());
    mDrawable.getPaint().setColor(color);
    mDrawable.setBounds(0, 0, size, size);
    mDrawable.setAlpha(alpha);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

Shapes, programmatically

In the Activity's onCreate():

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

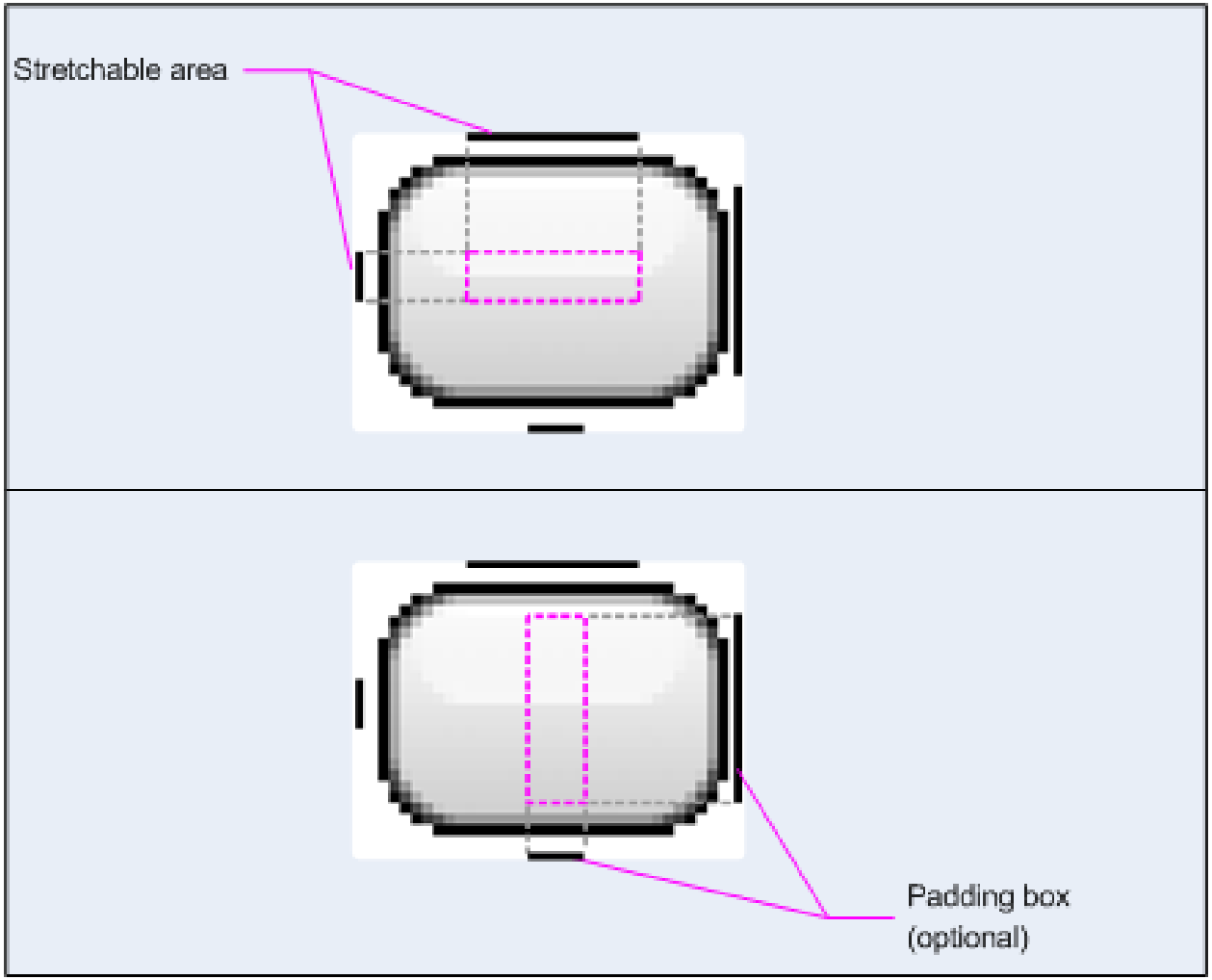
9-patches



I manually stretched the boxes using a graphics editor.

9-patches: automatic stretching

`NinePatchDrawable` can stretch your images automatically!



Just use a `.9.png` file. Edit using `tools/draw9patch`.