Lecture 13 — Android III

Patrick Lam
p.lam@ece.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 29, 2014

Part I

Android Persistent Storage

Storing Data

So far, we haven't seen any ways to store data so that you can re-load it across different instances of your application.

Four+ options:

- Shared Preferences
- Files: Internal and External Storage
- SQLite
- on the Internet

Shared Preferences

Key-value pairs.

Persists across user sessions, even if app gets killed.

Can be private to your application or shared across applications.

Using Shared Preferences

How do we get our hands on the shared preferences?

```
SharedPreferences settings = getPreferences(0);
```

Or, call `getSharedPreferences()` and pass it a preferences file name as the first parameter.

Reading Data from Shared Preferences

Now that you have a `SharedPreferences` object, you can get data from it:

```
v = settings.getString("textFieldValue", "");
```

Retrieves the value of `String` preference `textFieldValue`.

If no value is present, the `getString` call returns the default value `""`, an empty string.

Writing to Shared Preferences

It's a bit more complicated to write to the `SharedPreferences`.

You have to first get a `SharedPreferences.Editor`.

Then "commit" it once you're done with the changes.

```
SharedPreferences settings = getPreferences(0);
SharedPreferences.Editor editor = settings.edit();
editor.putString("textFieldValue", newFieldValue);
editor.commit();
```

Files: Internal & External Storage

Android phones have internal storage (small) and external storage (larger).

The commands to write to them are more or less the same.

Internal Storage

All Android devices have internal storage.

It is generally invisible to the user, other apps, and when the phone is mounted as USB Mass Storage.

When you remove an app, Android automatically erases its internal storage.

External Storage

External storage may be on an SD card (as on many phones).

It may be enclosed in the device (as on the Nexus 7 tablet).

Applications may access their own external storage space, shared storage space, or even other apps' storage space.

However, external storage may go away anytime, since the user could just pull out the SD card from the slot.

Files: External Storage

Our Lab 4 mapper code loads files from external storage in `MapLoader.java`.

You can use DDMS to put files into external storage, as well as using USB Mass Storage.

Files in external storage are world-accessible.

Warning: these files may go away anytime.

Files: Checking External Storage Availability

Sometimes the media is not there, or not writable.

```
String state = Environment.getExternalStorageState();

if (Environment.MEDIA_MOUNTED.equals(state)) {
    // We can read and write the media
} else if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) {
    // We can only read the media
} else {
    // Something else is wrong.
    // It may be one of many other states, but all we need
    //  to know is we can neither read nor write
}
```

Files: How MapLoader Works

It actually delegates to an XML parsing library.

Recall that you call:

```
    NavigationalMap map =
       MapLoader.loadMap(getExternalFilesDir(null),
                         "Lab-room-peninsula.svg");
    mapView.setMap(map);
```

The `MapLoader` contains this line:

```
File map = new File(dir, filename);
```

and then it builds a parser using the library call:

```
doc = docBuilder.parse(map);
```

which does all the I/O.

Files: about getExternalFilesDir()

Android provides a number of shared directories:

- DIRECTORY_MUSIC
- DIRECTORY_PICTURES
- DIRECTORY_RINGTONES

which all apps on the system can access.

Reading from Storage Example

The following code will read a line of text into the `String` variable `i`.

It currently reads from internal storage.

To read from external storage:
   1) comment out the `openFileInput` line; and
   2) uncomment the `new FileInputStream` line

Reading from Files

```
try {
    // internal storage:
    FileInputStream os = openFileInput("internal.txt");

    //InputStream os = new FileInputStream(
           new File(getExternalFilesDir(null), "external.txt"));

    BufferedReader br = new BufferedReader(
           new InputStreamReader(os));
    String i = br.readLine();
    // --> i contains the line you just read.
    os.close();
} catch (IOException e) {
   // Handle Exception...
}
```

Writing to Files

```
try {
    // internal storage:
    FileOutputStream os = openFileOutput("internal.txt",
                    Context.MODE_PRIVATE);

    //FileOutputStream os = new FileOutputStream(
                new File(getExternalFilesDir(null),
                    "external.txt"));

    PrintWriter osw = new PrintWriter(
                new OutputStreamWriter(os));
    // --> write out the contents of string i.
    osw.println(i);
    osw.close();
} catch (IOException e) {
    // Handle Exception
}
```

SQLite

SQL: Structured Query Language,
allows access to databases.

What’s a database?

id	front	frontFileName	back
17	“one”	“one.png”	“1”
42	“two”	“two.png”	“2”
99	“three”	“three.png”	“3”

Basic interface: queries on the database.

Part II

Digression: Labs

Lab 2: Sensor Data

This lab is about interpreting sensor data.

In particular, you are going to read the accelerometer data and count the number of steps taken by the holder of the phone.

There are two main problems:
1) sensor data is noisy; and
2) you need to recognize when a step occurs.

Sensor Noise

To deal with sensor noise, smooth the data.

Below is a very simple low-pass filter:

```
smoothedAccel += (newValue - smoothedAccel) / C;
```

Recognizing a Step

To recognize a step, you'll need to identify a change in the value of the *y*-axis acceleration.

Identifying a change means that you'll need the previous value along with the current value.

You could do that using a finite state machine (as described in the lab manual).

You could do that by doing tests both on the previous value and the current value.

Can you think of any pattern that might describe a step?

Part III

Toast, Broadcast Receivers, & Lists

Toast!

Sometimes you want to display a short message to the user. Use `Toast` to do that. Just include the following code:

```
Toast.makeText(getApplicationContext(),
               "A Toast!",
               Toast.LENGTH_LONG).show();
```

It’s also OK to use in your onClick event listener:

```
Toast.makeText(MainActivity.this, "A Toast!",
               Toast.LENGTH_LONG).show();
```

Broadcast Receivers

We talked earlier about the concept of Android Intents.

Android also uses Intents to broadcast information about what’s happening on the system between applications.

Applications want to know about events such as the phone being plugged into a power source; or, screen rotation.

Broadcast Analogy

A “party line” for Android Broadcast Receivers.

Many different applications can register an event listener, and they all get notified whenever something happens.

Broadcast Receiver Example: Rotation Listener

```
BroadcastReceiver broadcastReceiver = new BroadcastReceiver() {
   @Override
   public void onReceive(Context c, Intent i) {
     int orientation = c.getResources().getConfiguration().orientation;
     if (orientation == Configuration.ORIENTATION_PORTRAIT) {
       Toast.makeText(c, "Portrait", Toast.LENGTH_SHORT).show();
     } else if (orientation == Configuration.ORIENTATION_LANDSCAPE) {
       Toast.makeText(c, "Landscape", Toast.LENGTH_SHORT).show();
     } else {
       Toast.makeText(c, "???", Toast.LENGTH_SHORT).show();
     }
   }
 };
```

Register & Unregister the Receiver

To register: `registerReceiver(broadcastReceiver, intentFilter);`

To unregister: `unregisterReceiver(broadcastReceiver);`

Unfortunately, by default Android destroys your app on a screen rotate event.

This example only works if you tell Android not to do that.

Broadcast Receiver: Phone Ring Listener

This time to listen for phone calls.

First, you need to modify the `manifest.xml` file to permit the app to listen for phone calls:

```
<uses-permission android:name="android.permission.READ_PHONE_STATE">
</uses-permission>
```

Registering it in the Manifest

Include inside the `<application>` tag:

```
<receiver android:name="ca.patricklam.ece155demo.MyPhoneReceiver" >
    <intent-filter>
        <action android:name="android.intent.action.PHONE_STATE" />
    </intent-filter>
</receiver>
```

Phone Receiver

This means that we have to create a separate class MyPhoneReceiver:

```
public class MyPhoneReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    Bundle extras = intent.getExtras();
    if (extras != null) {
      String state = extras.getString(TelephonyManager.EXTRA_STATE);
      Log.w("PHONE", state);
      if (state.equals(TelephonyManager.EXTRA_STATE_RINGING)) {
        String phoneNumber = extras.getString
                                (TelephonyManager.EXTRA_INCOMING_NUMBER);
        Log.w("PHONE", phoneNumber);
      }
    }
  }
}
```

Lists

Another useful UI element is the `ListView`.

We can use it to show a list of items to the user (for instance, so that the user can choose one of the list elements).

There are a couple of caveats with using the `ListView`.

Creating a ListView

First, create a `ListView` object by dragging it onto the Activity's XML file.

- Note: you have to manually edit the `android:id` attribute so that its value is `@android:id/list`.

Next, change your Activity to be a ListActivity.

Add a field `listAdapter` of type `ArrayAdapter<String>`.

Populating a ListView

Populate the list with entries:

```
List<String> data = new ArrayList<String>();
data.add("ECE155");
data.add("ECE106");
data.add("ECE124");
listAdapter = new ArrayAdapter<String>(this,
                         android.R.layout.simple_list_item_1,
                         data);
setListAdapter(listAdapter);
```

Clicking an Item

Finally, we want something to happen when we click on a list item:

```
@Override
protected void onListItemClick(ListView l, View v,
                                  int position, long id) {
  super.onListItemClick(l, v, position, id);
  String s = (String) getListAdapter().getItem(position);
  Toast.makeText(this, "Aha: "+s, Toast.LENGTH_SHORT).show();
}
```

Display a toast when the user chooses a list item.

Dynamically Updating a ListView

Of course, we can also add items to the `ListView` programmatically. In a click listener (or anywhere else), you can write:

```
String now = String.valueOf(System.currentTimeMillis());
listAdapter.add(now);
```

ListView

Note that adding elements to the ArrayAdapter also adds them to the ListActivity.

Also, we currently store the ArrayAdapter as a field.

That means it'll go away whenever the Activity is destroyed (e.g. rotation).

We'll want to do something about that.