Advanced Android

CSCI 4448/5448: Object-Oriented Analysis & Design
Lecture 19 — 10/30/2012

Goals of the Lecture

- Present more examples of the Android Framework
  - Passing Information between Activities
  - Reading and Writing Files
  - 2D Graphics and Touch Events
  - Application Preferences
  - Working with a Database

Passing Information

- In our examples so far
  - we've seen one activity launch another activity
  - but each activity has been independent of the other
- We're going to look at two additional concepts
  - Fragments: reusable bits of UI and behavior that live inside activities
  - Passing Information: how do we pass information between activities
    - We'll also take a look at how an activity can store data into a file that persists between sessions of using the application

Profile Viewer

- Profile viewer will
  - Use one activity/fragment to display a list of user names
    - This activity can also delete existing users
  - Use a second activity/fragment to add new users and edit existing users
- Our program **will use Java serialization to persist user names and profiles**
  - The data structure will be a Map<String, ProfileData>
  - We'll discuss ProfileData in a moment
- But first, fragments!

Activities and Fragments (I)

- Activities can now contain multiple fragments
  - Fragments are reusable units of UI with their own life cycle
    - this life cycle is however synched with the life cycle of its activity
      - onCreate(), onPause(), onResume(), etc.
  - Fragments provide flexibility when presenting the UI of an application on either a phone or a tablet
    - You're initial activity can detect how much screen real estate is available and then either choose to display fragments in a set of activities (for phones with small displays) or to embed multiple fragments inside of a single activity (for tablets with larger displays)
- Migrating to fragments from activities is straightforward

Activities and Fragments (II)

- The scenario I outlined on the previous slide is shown graphically here:

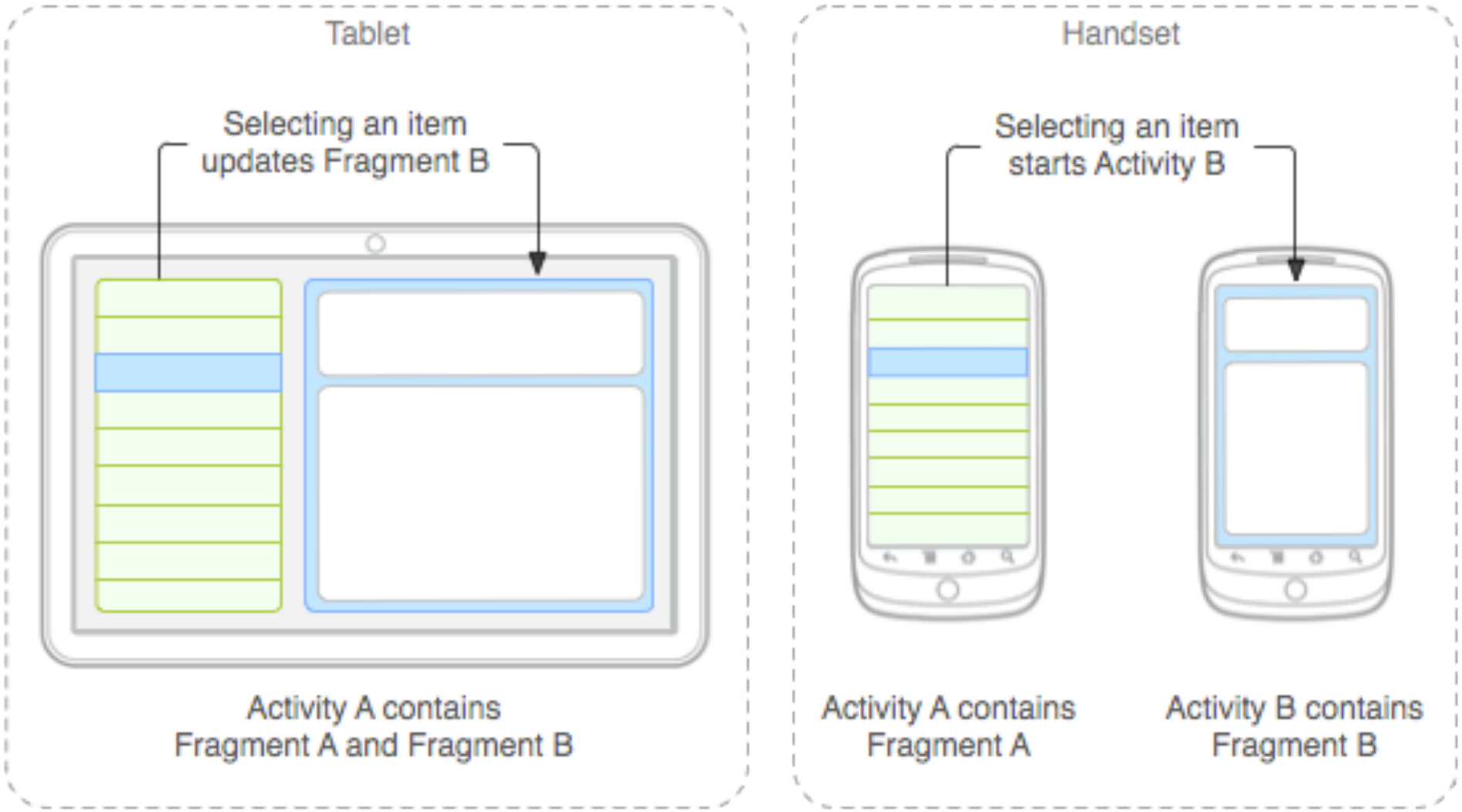

Image credit: <http://developer.android.com/guide/components/fragments.html>

Java Serialization (I)

- Java serialization is a technology that can both
  - persist a set of objects, and
  - later retrieve that set such that all objects are recreated and all connections between them are reestablished
- **java.io** provides two classes to help with this
  - **ObjectOutputStream** and **ObjectInputStream**
  - You use the former to save and the latter to load

Java Serialization (II)

- Most Java types, including collections, can be serialized
- User-defined types can also be serialized
  - You need to implement **java.io.Serializable**
  - And, you need to implement two methods
    - **readObject(ObjectInputStream stream)**
    - **writeObject(ObjectOutputStream stream)**

Java Serialization (III)

- In **writeObject()**, you place code that writes each internal attribute of your object on to the output stream
- In **readObject()**, you place code that reads each attribute off of the input stream in the same order they were written by writeObject
- Then, when it comes time for your class to be persisted, Java's serialization framework will call readObject() and writeObject() as needed passing the appropriate IO stream

ProfileData (I)

- For our Profile Viewer application, our ProfileData class stores a user's first name, last name, and e-mail address
    - ProfileData is implemented as a data holder with getter and setter methods for each attribute
- It implements **java.io.Serializable** as needed
- It also contains a **serialVersionUID** attribute—generated by Eclipse—that is used to add support for versioning.
    - If we ever change the ProfileData class, we'll need to update the UID.
        - Advanced implementations would then use the UID to determine which version a file used and load it using the correct code

Profile Data (II)

- Our writeObject Method looks like this

```
private void writeObject(ObjectOutputStream stream) throws
IOException {

      stream.writeObject(firstName);

      stream.writeObject(lastName);

      stream.writeObject(email);

}
```

- writeObject() is defined multiple times for multiple types

Profile Data (III)

- Our readObject Method looks like this

```
private void readObject(ObjectInputStream stream) throws
IOException, ClassNotFoundException {

      firstName = (String)stream.readObject();
      lastName  = (String)stream.readObject();
      email     = (String)stream.readObject();

  }
```

- If we try to read a String and the file contains something else, then a ClassNotFoundException will be thrown by ObjectInputStream

Java Serialization (IV)

- Having configured ProfileData in this way, then the code to write a Map<String, ProfileData> data structure is:

```
ObjectOutputStream output =

  new ObjectOutputStream(new FileOutputStream(f));
output.writeObject(profiles);
```

- Two lines of code! (Ignoring exception handlers)

Java Serialization (V)

- The code to read a Map<String, ProfileData> is:

```
ObjectInputStream input =
  new ObjectInputStream(new FileInputStream(f));
profiles = (TreeMap<String,ProfileData>) input.readObject();
```

- Just two more lines of code!

Java Serialization (VI)

- Hiding in those two lines of code was a reference to a variable named “f”; Here’s the relevant part:
  - **new FileInputStream(f)** or **new FileOutputStream(f)**
    - As an aside: **java.io** is based on the **Decorator** pattern
- In both cases, we were passing an instance of **java.io.File** to the IO streams to specify where our persistent data is stored
- So, now we need to look at how we deal with files in Android

Dealing With Files (I)

- Each Android application has a directory on the file system
  - You can verify this by launching an emulator and then invoking the “adb -e shell” command
    - adb is stored in $ANDROID/tools (2.x) or $ANDROID/platform_tools (3.x and 4.x)
  - This command provides you with a command prompt to your device; recall that Android runs on linux
    - cd to **data/data** to see a list of application directories
  - If you encounter permission problems, run the command “su” and see if that helps
    - If so, be careful, you’re now running as root!

Dealing With Files (II)

- For Profile Viewer, cd into the **edu.colorado.profileviewer** directory (you'll need to compile and install Profile Viewer onto your device first!)
  - That directory contains two subdirectories
    - **files** and **lib**
      - Whenever you ask for access to your application's directory and create a file, it will be stored in the "files" subdirectory
- Application directories are private; other apps can't access them

Dealing With Files (III)

- Android provides several useful methods for accessing your application's private directory
  - **getFilesDir()** - returns a java.io.File that points at the directory
  - **fileList()** - returns list of file names in app's directory
  - **openFileInput()** - returns FileInputStream for reading
  - **openFileOutput()** - returns FileOutputStream for writing
  - **deleteFile()** - deletes a file that is no longer needed

Profile Viewer's Use of Files

- In Profile Viewer, all we need to use is **getFilesDir()**
  - We use that to create a java.io.File object that points at a file called "profiles.bin" in our app's directory
  - We then pass that file to our save/load methods
  - That code looks like this
    - **profiles.load(new File(getFilesDir(), "profiles.bin"));**

Back to “Passing Information”

- When we select a user and click Edit, we switch from the initial activity to an “edit profile” activity
  - We want that second activity to display the profile of the selected user
    - How do we pass that information?
      - In Android, that information gets passed via the Intent that is used to launch the second activity

Passing Information (II)

- Each intent has a map associated with it that can store arbitrary Java objects
  - The Map is updated via **putExtra(key, value)**
  - The Map is accessed via **get*Extra(key)** where "*" can be one of several type names
    - In Profile Viewer, we use **getStringExtra(key)** because the user name we store is a string
- An activity can get access to the intent that launched it via a call to **getIntent()** which is a method inherited from Activity

Passing Information (III)

- So, to pass information we do this in our fragment
  - `Intent intent = new Intent(this, EditProfile.class);`
  - `intent.putExtra("name", username);`
  - `getActivity().startActivity(intent);`
- To retrieve it, we do this in the Edit Profile fragment
  - `username = getActivity().getIntent().getStringExtra("name");`
- Simple!

Other Highlights

- Profile Viewer also shows
    - how to use fragments and how they interact with activities
    - how to add menu items to the ActionBar
        - how to enable/disable menu items based on list selections
        - how to save/load data in onResume() and onPause() to ensure that data is synced between activities

Demo

2D Graphics and Touch Events

- The Simple Paint program takes a look at how to do simple 2D graphics in Android
    - and how to handle touch events
- Whenever you want to do your own drawing, you need access to a canvas
    - If you create a subclass of View and then override the onDraw(Canvas) method, you gain access to a canvas
    - Essentially, a view IS-A canvas

Key Concepts (I)

- We draw on a canvas
  - In order to draw a shape, we first need a Paint object; it specifies a wide range of attributes that influences drawing
  - We then invoke one of canvas's draw methods, passing in the shape info and our paint object
- In our program, we create one Paint object called background which we use to paint the canvas white
  - and a second Paint object used to paint Rectangles

Key Concepts (II)

- Draw on Demand
  - As with most frameworks, drawing in Android is done on demand when the framework determines that an update is needed
    - say if our view gets exposed because a window on top of it moves
    - or when our own code calls invalidate()
- onDraw is then called and we draw the current state of the view as determined by our program's data structures
  - onDraw() is where all drawing occurs; it does NOT occur (for instance) when we are handling a touch event
    - This is an important concept, the event handler for touch events simply updates our data structures and returns; drawing happens later

OnDraw (I)

- Our SimplePaint program allows rectangles to be drawn in four different colors
- We have a data structure that keeps track of the rectangles that have been created and the Paint object used to draw each one
    - If we are in the middle of handling a touch event, a rectangle called motionRect exists and we will draw it as well
- Our onDraw method is shown on the next slide

OnDraw (II)

```
protected void onDraw(Canvas canvas) {
  canvas.drawRect(0, 0, getWidth(), getHeight(), background);
  for (Rectangle r : rects) {
     canvas.drawRect(r.r, r.paint);
  }
  if (motionRect != null && motionRect.bottom > 0 &&
 motionRect.right > 0) {
     canvas.drawRect(motionRect, current);
  }
}
```

Handling Touch Events (I)

- To handle a touch event on our custom view
  - we override the **onTouchEvent()** method
  - we then process the MotionEvent instance that we are passed
  - and then return true to ensure that we get all of the events related to the touch event
- There are three stages:
  - DOWN (the start), MOVE (updates), UP (the end)

Handling Touch Events (II)

- An ACTION_DOWN event means that the user has just touched the screen
  - In our program, we create motionRect and set its top, left corner
- An ACTION_MOVE event means the user is moving their finger across the screen
  - we update the bottom, right corner and invalidate
- An ACTION_UP event means the user has lifted their finger from the screen
  - We update motionRect with the last x, y coordinate, add motionRect to our data structures and then set motionRect to null

Handling Touch Events (III)

- Finally, to actually receive touch events, we need to do three things
  - In the constructor of our View subclass, we need to call
    - **setFocusable(true);**
    - **setFocusableInTouchMode(true);**
  - In the constructor of our activity, we get a handle to our View subclass and call **requestFocus()**;
    - That ensures that Android sends events to the view

Other Highlights

- Simple Paint also demonstrates the use of
  - a radio group to keep track of the current paint color
  - Android's preference mechanism to let the current paint color persist between runs of the application
    - You call getSharedPreferences to gain access to a map that contains your apps preferences
      - You can read and write preference values in a straightforward manner

Demo

Android's support for SQLite

- Android makes it straightforward to interact with SQLite databases
  - SQLite is a public domain SQL library that stores a database as a text file and provides standard CRUD operations on that text file
    - as if you were actually talking to a database server
- Android provides a class to make creating/opening a database a snap, a class that allows standard select, insert, update and delete statements to be executed and a Cursor class for processing result sets

SQL Example

- For this example, I recreated Profile Viewer and
  - dropped our custom Profiles / ProfileData classes that made use of Java serialization
  - and incorporated the use of an SQLite database
- As you will see, all of the original functionality could be recreated and the resulting program is just a tad simpler
  - **IF** you are comfortable with database programming and SQL; if not, it will seem confusing!
- Note: this version of the program does not use Fragments
  - To keep things simple, this program only uses activities to handle the UI

SQLiteOpenHelper

- To create a database, you make a subclass of SQLiteOpenHelper
  - It takes care of creating and opening a SQLite database for you at run-time
  - All you need to do is to supply the CREATE TABLE statement needed to create the table you'll be using
  - I created a table whose columns correspond to Profile Viewer's profile name, first name, last name, and e-mail address attributes

Accessing the Database

- In your activity, creating an instance of your OpenHelper subclass, automatically creates (if needed) your database and opens it
    - In your onStop() method, you need to remember to close the database
- You then can acquire the database for reading or writing as needed with calls to getReadableDatabase() or getWriteableDatabase()

CRUD Support

- In databases, you can create, read, update or delete rows in a table
    - In Android's database object these correspond to
        - insert, query, update, delete
- These are methods, you supply snippets of SQL to these methods; they create the full SQL statement in the background and then execute it against the database

Selected Snippets (I)

- Getting a list of profile names from the database
  - SQLiteDatabase db = profileDB.getReadableDatabase();
  - Cursor cursor =
    - db.query("profiles", new String[] { "profile" }, null, null, null, null, "profile");
- while (cursor.moveToNext()) {
  - adapter.add(cursor.getString(0));
- }
- cursor.close();

Selected Snippets (II)

- Deleting a profile from the database
    - SQLiteDatabase db = profileDB.getWritableDatabase();
    - db.delete("profiles", "profile = ?", new String[] { name });
- The “profile = ?” is part of an SQL WHERE clause;
    - the ? mark is a placeholder
    - It gets replaced by the value of the variable “name” which is passed in via a String array: “new String[] { name }” is a string array literal in Java

Selected Snippets (III)

- Inserting a new profile into the database
- SQLiteDatabase db = profileDB.getWritableDatabase();
- ContentValues values = new ContentValues();
- values.put("profile", name);
- values.put("first", first);
- values.put("last", last);
- values.put("email", email);
- db.insertOrThrow("profiles", null, values);

Wrapping Up

- Learned more about the Android framework
  - Passing Information between Activities
  - Reading and Writing Files
  - 2D Graphics and Touch Events
  - Application Preferences
  - Working with a Database
- This ends our woefully incomplete review of the Android Framework; however, our three lectures should be enough to get you started!

Coming Up Next

- Lecture 20: Advanced iOS
- Homework 4 Due Next Week