Mobile User Interfaces

CS 2046
Mobile Application Development
Fall 2010
Announcements

• **Next class = Lab session: Upson B7**

• **Office Hours (starting 10/25):**
  – Me: MW 1:15-2:15 PM, Upson 360
  – Jae (TA): F 11:00 AM-12:00 PM, Upson 328B, Bay D
Recap

• Task = Stack of related Activities

• Activity Lifecycle:
  – Active = In the foreground, called onResume()
  – Paused = Obscured, called onPause()
  – Stopped = Not visible, called onStop()

• Resources – separate program logic from “other stuff”
  – Strings, images, UI layouts

• AndroidManifest.xml – tie together components
Processes/Threads on Android

• Default: One application = one process
  – All components are instantiated on main thread
  – Consequence: Should not perform long/blocking operations without forking.

• Definition of application: all components grouped under the <application> tag in AndroidManifest
Service Lifecycle

• Service has `onCreate`, `onStart`, and `onDestroy` methods (but no pause/resume).

• `startService` (like `startActivity`) will start the service if it isn’t already running.

• Then, call `onStart`
  – If service is already running, just call `onStart`

• As consequence of single process, should spawn new Thread to handle work.
Service.stopSelf

• `onStart(Intent i, int startId)`

• Should stop Service when all commands have been processed.
  – Because of threading – no guarantee of order.

• `stopSelf(startId)`: Will stop Service if `startId` corresponds to last command.

• What data structure should we use to store `startIds`?
Service


- Also see my post on the Newsgroup

- We’ll talk about this more in a few lectures when we get to background tasks.
Developer Blog – Improving Apps

• Listen to your users
  – Use betas before making final releases

• Improve stability and eliminate bugs
  – Monkey: Send random UI events
  – View reported crashes in Android Market

• Improve UI Responsiveness
  – Today’s lecture!

• Integrate with the system and third-party apps

• See post for many more:
Designing Mobile Interfaces

• Reading: 12 myths of mobile UI design
  – (Most of these are not actually related to UI)

• Takeaways:
  – Keep it simple!
  – Get in the shoes of your users
    • Figure out what 90% of your users need to do 90% of the time, and make it easy.
  – Consistency is key.
    • The less you reinvent the wheel and use standard UI features, the more your user will already know how to use.
View Hierarchy

• View: Basic unit of user interface
  – Widgets: android.widget.*
  – Leaves of the hierarchy tree

• ViewGroup: Defines a layout
  – Also in android.widget.*
  – Define where to place children Views (or ViewGroups)
Examples of Widgets

- Button

- EditText

- CheckBox and RadioButton

- Spinner

- Others: TextView, ImageView

- Can extend a widget, or create a brand new one.
Examples of Layouts

• Remember: ViewGroup is just a subclass of View

• Simplest: FrameLayout
  – Blank space to be filled with a single object
  – Pins objects to top-left corner
  – If it contains more than one object, just draws them overlapping

• This is the root layout for any Activity
LinearLayout

• Aligns all children one after the other in a single direction
  – orientation = vertical or horizontal
• Can nest for more complex layout.
  – Common use: (small) forms
  – Vertical LinearLayout
    • Horizontal LinearLayout
      – TextView = Label
      – EditText = Value
    • Horizontal LinearLayout
      ...
TabLayout

• Multiple distinct features in one View
• Good use of constrained screen space
• Two methods of filling tabs:
  – Swapping Views – best for features which are similar and fit into one functional group.
  – Swapping Activities – best for managing separate tasks, instead of one massive Activity and layout.
Other Layouts

• **RelativeLayout**
  – Specify elements relative to parent or siblings
  – Combination of power and simplicity in description

• **TableLayout**
  • EditText: Below TextView
  • OK Button: Below EditText, aligned to right of parent (Screen)
  • Cancel Button: Left of OK Button, small margin to the right
Nested LinearLayouts

• Nesting LinearLayouts is an easy way to create many common UIs.

• Caution: doing this too much (more than ~5 levels of nesting) can make UI sluggish!
LinearLayout → Relative Layout

- Solution: Refactor into a RelativeLayout
  - Example:

- Might not seem like much, but if this is drawn on screen many times...
ListView

- ViewGroup containing a list of Views
- Can define a View to display when List is empty with setEmptyView
- Each row is by default just a TextView – can customize.
- Usually populated dynamically
  - Assignment 1: String[] array of tasks
  - Assignment 2: Data pulled from a ContentProvider.
ListView Adapters

• Adapter – Binds dynamic content to the Views in a ListView.
  – e.g. ArrayAdapter for arrays

• Simple – bind a text value to the one text field in the ListView.

• More complex – custom ListView rows, custom objects being bound to the Views.
  – We’ll see this in Assignment 2
Layout Weight

- Weight allows creating LinearLayouts with proportional sizes.
- Default = 0 – minimum space to display all content
Example

• How would you define this layout?
Defining Layout

• Most common method – XML layout files
• Located in res/layout/<file>.xml – can access later as R.layout.<file> from code.

• All files contain:
  – XML version:
    
    ```xml
    <?xml version="1.0" encoding="utf-8"?>
    ```
  
  – xmlns:android tag in root element
    
    ```xml
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android">
    ```

• From onCreate, call setContentView(R.layout.<file>) to set the root layout element for an activity.
XML Layout

• All attributes preceded with “android:”

• Attributes which apply to every View:
  – id – optional – a unique identifier for the object so it can be accessed later from code.
    • Specify as android:id="@+id/<name>"
    • In Java: (Button) b = (Button) findViewById(R.id.<name>);
  – layout_width/layout_height – dimensions of object
    • Specific size (pixels, or dips – density-independent pixels)
    • fill_parent: Takes up entire size of the parent ViewGroup
    • wrap_content: Takes up only as much space as is needed to display the View.
Styles

• Essentially, CSS for Android UI
• Located in res/values/anything.xml
  – styles.xml is a good choice

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="BigText">
    <item name="android:textSize">30dip</item>
  </style>
</resources>
```

• Apply to a view in XML with:
  style="@style/BigText"
Themes

- Theme: Style applied to entire Activity or application
- Apply in AndroidManifest.xml

- Common examples:
  - Make Activity look like dialog box:
    <activity android:theme="@android:style/Theme.Dialog">
  - Get rid of title bar:
    @android:style/Theme.NoTitleBar

- More on Styles/Themes:
Programmatic Layout

• Can also define layout in Java:

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    LinearLayout ll = new LinearLayout(this);
    Button button1 = new Button(this);
    button1.setText("Hello");
    Button button2 = new Button(this);
    button2.setText("World");
    ll.addView(button1);
    ll.addView(button2);
    setContentView(ll);
}
```

• In practice, much easier to define XML layout
  – But, need Java to receive events from Views and update them dynamically.
More on UI

• Resources:
  – Hello, Views tutorial:
    • Examples (with code) for these and other layouts
    • Examples for common widgets (Date Picker, Spinners, Image Galleries, etc.)
  – Javadoc for android.widget package:
    • For each View/ViewGroup, describes:
      – XML attributes that can be specified
      – Methods that can be called on the Java object.
  – Official developer blog posts on UI:
    http://android-developers.blogspot.com/search/label/User%20Interface
  – API Demos
UI in Java

• Have defined layout in XML – now what?

• We need to know how to:
  – Get data from and put data to widgets
    • How do we access the text in an EditText?
  – Receive events from widgets
    • How can we do something when a button is clicked?
Getting References

- Have view with `android:id="@+id/widget"`
  - `<Class> widget = (<Class>) findViewById(R.id.widget);`
    - `<Class>` is the class of the View, i.e. Button or EditText
    - This is the object we need to:
      - Get/set fields
      - Set up event handlers
Gotchas

• If you need to access a view in more than one method:
  – Have a class variable prefixed with m (for member):
    • private ListView mList;
  – Bind variable in onCreate() method

• What’s wrong with this:

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Button b = (Button) findViewById(R.id.button);
    setContentView(R.layout.main);
}
```
Gotchas

• If you need to access a view in more than one method:
  – Have a class variable prefixed with m (for member):
    • private ListView mList;
  – Bind variable in onCreate() method

• What’s wrong with this:

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Button b = (Button) findViewById(R.id.button);
    setContentView(R.layout.main);
}
```
Gotchas

• If you need to access a view in more than one method:
  – Have a class variable prefixed with m (for member):
    • private ListView mList;
  – Bind variable in onCreate() method

• You must first call setContentView() before findViewById will work!
Getting/Setting Fields

• Once we have a reference to a widget, accessing fields is fairly simple:

  • EditText textField = ...
  String text = textField.getText().toString();
  textField.setText(“Hello”);
  textField.setTextColor(Color.RED);
Interlude – Anonymous Inner Classes

• Consider the (generic) Java method: `Arrays.sort(T[] a, Comparator<T> c)`

• Comparator interface:
  – `int compare(T o1, T o2)`
    • Return 0 if equal, -1 if `o1 < o2`, 1 if `o1 > o2`

• How should we construct an instance of Comparator?
Anonymous Inner Classes

class Pair {
    int x, y;
    boolean equals(Object other) { ... }  
...
}

class PairComparator implements Comparator<Pair> {
    int compare(Pair p1, Pair p2) {
        if (p1.equals(p2))
            return 0;
        if (p1.x < p2.x || p1.x == p2.x && p1.y < p2.y)
            return -1;
        else
            return 1;
    }
}

Arrays.sort(Pair[] pairs, new PairComparator());
Anonymous Inner Classes

class Pair {
    int x, y;
    boolean equals(Object other) { ... } ...}

Arrays.sort(Pair[] pairs, new Comparator<Pair>() {
    int compare(Pair p1, Pair p2) {
        if (p1.equals(p2))
            return 0;
        if (p1.x < p2.x || p1.x == p2.x && p1.y < p2.y)
            return -1;
        else
            return 1;
    }
});