Android Overview: Intro to Activities

Android Overview

- You‘ve already done some Android programming.
- This material may not be new, but it‘s worth reviewing.
- Android apps are typically made up of upto 4 connected components:
  1) Activities.
  2) Services.
  3) Broadcast receivers.
  4) Content providers.

Android Overview

- **Activities:** a single screen with a user interface (UI), the entry point for user interaction. Activities have to do a lot.
  - Manage the lifecycle of the app, make sure things important to the user are kept around and not killed/stopped by the OS.
  - Help manage user flows between apps, possibly via intents, and within a given app.
- **Services:** no UI. Typically runs in the background for long-running tasks, performs work for remote processes. Typically started by an activity. Eg. Keep playing that Spotify song even if you‘ve switched to some other app.

Android Overview

- **Broadcast receivers:** components that allow the app to receive events from the OS that are *not* a part of the regular function of the app.
- **Content providers:** manage shared app data to be stored either locally on the phone, on a remote database, or some persistent storage location. Eg. Contact information is managed by a content provider, can be used by other apps if they have permission.
- We‘ll focus on activities for now, but we‘ll discuss all these components, and how to program/use them.

Activities

Activities

- Let‘s discuss this from a coding perspective.
- We‘ll use Android studio and Kotlin.
- We‘ll focus for now on developing the UI used in activities, and in working efficiently with multiple activities.
- Later, we‘ll discuss the lifecycle of an activity and how it is managed by the OS.
- We‘ll also discuss user data persistence later.
- So, what is an Activity to a programmer?

Activities

- Each screen/page in your app is an Activity.
- Each Activity involves at least a pair of files: a .java or .kt file, and a corresponding .xml file.
- The .kt file for an activity contains a subclass of the Activity class. For instance, **MainActivity** almost always extends **AppCompatActivity**, which in turn extends **FragmentActivity**.
- You override specific functions of the base class that you‘re extending to get different kinds of functionality.
- For instance, if you override onCreate(), you get to do stuff right when the app is created. If you override onDestroy(), you handle cleanup when the app is destroyed.

Activities

- The function names hint at some sort of lifecycle for an Activity, and for an app. More later.
- What goes in the .xml file? How does the .xml file interact with the .kt file?
- The .xml file contains our descriptions of UI elements.
- The .java file is responsible for **inflating** the .xml file, and then managing events related to that UI.
- **Inflating** in this context means
  - First parse the .xml file
  - Create Java objects corresponding to the .xml file.
  - Provide the programmer means to interact with the Kotlin objects and therefore with the UI (using the **R** class).

Activities

- Let‘s look at a simple .xml file and its .kt mate.
- Included here for convenience. But we‘ll look at code.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

```
<android.support.constraint.Frame
Layout
xmlns:android="http://schemas.and
roid.com/apk/res/android"
xmlns:app="http://schemas.android
.com/apk/res-auto"

android:layout_width="match_paren
t"

android:layout_height="match_pare
nt">

    <TextView

android:layout_width="wrap_conten
t"

android:layout_height="wrap_conte
nt"
        android:text=""
        android:padding = "16dp"
        android:textSize="20sp"
        />

</android.support.constraint.Fram
eLayout>
```

Activities

- The xml is in the activity_main.xml file.
- The Kotlin code is in the MainActivity.kt file.
- We override onCreate() from AppCompatActivity.
- Using setContentView, we **inflate** the xml file, then render onto the screen.
- Inflation is done by traversing the xml **tree**.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Activities

- What does this mean for us, practically speaking?
- Better learn to manipulate xml directly.
- Android Studio also has a GUI (design view) for manipulating xml.
- Let‘s see how to develop a simple UI for an app.
- Simple process:
  - Sketch out the app, either by hand or in some software.
  - Identify elements of your sketch, and decide what Android *things* they correspond to.
  - Create the xml for your sketch.
  - Inflate the xml in your Java code.

Activities: Example 1

- Let‘s design a UI without any user interaction, but with a lot of elements on screen.
- Identify the elements on the screen.
- Are there any elements on screen that you can‘t see?
- How does one achieve this specific orientation of these elements?

Activities: Example 1

- **Identify the elements on the screen.**
  - The colored items are all TextView objects, all instances of the View object. This is the basic Android textbox.
  - How do you color them?
- **Are there any elements on screen that you can‘t see?**
  - There is a LinearLayout ViewGroup organizing all the TextViews in a nice column.
  - There is another LinearLayout ViewGroup organizing the fourth row into columns.

Activities: Example 2

- The TextViews are all bumping up against the text boxes.
- What if we want to separate them?
- We need to add whitespace to the views.
- Essentially two ways to add whitespace: padding, and margin.
- Attribute android:padding adds space **inside** a view or ViewGroup.
- Attribute android:layout_margin adds space **outside** a view or ViewGroup.
- Can you identify padding and margin in the following example?

Activities: LinearLayout

- Let‘s summarize.
- We saw the LinearLayout ViewGroup. All views must sit inside a ViewGroup of some sort.
- We saw that the LinearLayout can be either horizontal or vertical.
- For LinearLayout, the order of items in the xml file matters!
- We learned the difference between layout_margin and padding. The key is to also check layout_width and height.
- What if I wanted a single View?
- What if I wanted Views arranged in a grid?
- What if we wanted more flexibility in our layouts?

Activities: Example 3

- Now, let‘s see an example with a single View.
- If you only have a single View, you wrap it in a FrameLayout.
- Well, that was straightforward.
- But… what if I add more than one View to a FrameLayout?
- Does it break?
- What happened?

Activities: FrameLayouts do Z-ordering

- When you add multiple Views to a FrameLayout, they **stack**.
- The last added view is seen on top.
- This is called Z-ordering. Z here refers to the z-axis that is pointing into the screen.
- Using FrameLayouts, we can decide how *deep* some views are relative to others.
- This is very useful for overlaying text onto other views, such as images.

Activities: Grids

- Imagine you wanted a nice grid of Views.
- You could easily do this by nesting LinearLayouts. For instance:

```
<LinearLayout … android:orientation=vertical>
    <LinearLayout … android:orientation=horizontal>
    </LinearLayout>
</LinearLayout>
```

- What‘s the problem with this?
- It may take the OS a lot of **time** to traverse the tree corresponding to this view hierarchy.

Activities: Grids

- Fortunately, GridLayout exists for this sole purpose.
- Not a lot of developers seem to know about it.
  - It’s a little annoying to use sometimes.
- You can create a grid of views on the screen.
- Each view can span one or more grid cells.
- It‘s more efficient than a whole bunch of nested LinearLayouts.
- Best to learn by example.

Activities: Example 4

- GridLayout in this example.
- Notice how you can make a view span more than one row or column in the grid.
- Specify the number of columns up front.
- Don‘t worry about ordering. Android will arrange views for you.
- Grid cells are given a *linear* index. You start with top left, and count to find the index of the grid cell you‘re on.

Activities: RelativeLayout

- RelativeLayout can do everything we discussed with the other ViewGroup types, and more.
- As the name implies, it allows you to position child Views relative to the parent (the RelativeLayout itself).
- It also allows you to position child Views relative to each other!
- As you can imagine, far more flexible than LinearLayouts.
- The child Views can even be other ViewGroups.
- Allows you to develop elegant and potentially complex UIs.

Activities: Example 5

- This example shows a RelativeLayout.
- There are a whole bunch of things to pay attention to.
- Since Views are positioned relative to each other, you need a way to *identify* a view. For this, we use android:id=“stuff“.
- If you‘re *creating* an id, do android:id=“@+id/name_of_view“.
- If you‘re simply referring to an id, do android:id=“@id/name“.
- Note the alignment parameters wrt both the parent and child views.
- We also added a **weight** to the EditText. A larger value of layout_weight allows this EditText to expand to fill up any space in the LinearLayout.

Activities: Summary

- LinearLayouts allow for row/column arrangements of Views.
- FrameLayouts stack their views, allow for z-ordering.
- GridLayouts allow for grid arrangements of views.
- RelativeLayouts allow you to do all the above, and more.
- Clearly, RelativeLayouts are the most flexible and powerful.
- The Android default is ConstraintLayout.

Activities: Lab

- Let’s do a lab where you implement some of these views.
- Implement the UI for the sketch on Canvas.
- First, implement using LinearLayout.
- Next, implement using RelativeLayout.
- If you want, try and implement with GridLayout.