More Activities and User Interaction

Recap

- You learned about a few different ViewGroups: LinearLayout, FrameLayout, GridLayout, and RelativeLayout.
- You learned about padding, margin, and layout width, height, weight, and gravity.
- You compared these different ViewGroups on the same toy program.
- You also saw some Views: TextView, EditText, and Button.
- Now, we'll look at the most important ViewGroup: ConstraintLayout.
- We'll also start some event-driven programming.

Activities: ConstraintLayout

- This is RelativeLayout on steroids.
- What was the primary issue with RelativeLayout?
- It still potentially could suffer from nested ViewGroups and complex view hierarchies.
- The more complicated the view hierarchy tree, the more resources it will take to traverse that tree.
- ConstraintLayout was created to overcome this issue.
- How?

Activities: ConstraintLayout

- Much like RelativeLayout, you specify the locations of Views with respect to each other, and wrt to parent layout.
- However, ConstraintLayout does this by imposing **constraints** on the different views.
- The views are constrained to each other, and to the parent.
- Views can be constrained to **groups** of views, without explicitly using a ViewGroup to group them.
- The ViewGroup tree is flat since there are no real children of children.
- ConstraintLayout is harder to program, but sometimes very useful.

Activities: ConstraintLayout

- With RelativeLayout, you don’t really need to use the Design view in Android Studio.
- With ConstraintLayout, it is usually pretty hard to do things without Design view.
- I’ve found that my workflow for ConstraintLayout is:
  - Design UI using design mode.
  - Polish up measurements in text mode.
  - Repeat.
- Let’s see an example of developing a UI with ConstraintLayout.

Activities: Example 6

- I made the same UI as Example 5, except with ConstraintLayout.
- There are many possible constraints that would lead to the same arrangement of Views on the screen.
- I was able to do it mostly in design mode, but I had to switch back to xml a couple of times.
- You could also design a UI in RelativeLayout, and have Android Studio convert it to a ConstraintLayout for you.
- It will flatten the view hierarchy, remove any nested layouts, but achieve the same effect.
- Let’s see Android Studio’s solution and how it compares to mine.

Activities: Example 7

- Started with Example 5's RelativeLayout as a base, but asked Android to convert to ConstraintLayout.
- Compare to Example 6.
- The constraints are different.
- In fact, the output isn't exactly right.
- Lesson: don't rely entirely on an automated process for design.
- Design hinges on key elements of human perception, psychology, and our sense of aesthetics and spacing.
- These are often hard (but not impossible) to capture using algorithms.

Basics of User Interaction

User Interaction

- We’ve been creating static, lifeless user interfaces (UIs).
- The UI is the part of the app that faces the user.
- It needs to be able to accept and operate on input:
  - Handle EditText and its text input.
  - Handle button clicks.
  - Show messages to users alerting them about stuff.
  - Open a new screen based on some input.
  - Go back to the original screen.
  - Open some other app (maps, mail, camera, whatever).
  - Many more types of interaction...

User Interaction: Basics

- For now, we'll focus on the types of interaction from the previous slide.
- Over a series of examples, we'll create a single UI that is able to handle various types of input.
- Then, you'll do this in your lab session (and your HW, and your project).
- It's okay if you already know this stuff. We'll learn some of the "best practices" here.

User Interaction: Example 8

- Let’s enter a first name and last name in an EditText.
- We’ll read them, break them up, and display them in two TextViews.
- The TextViews will be populated by a button click.
- When the button clicks, we’ll also show one of two messages:
  - If the EditText is empty, then display a brief message asking them to enter text.
  - If the EditText is populated, we’ll say “Good work!”.
  - We’ll also have to handle edge cases.
  - A “Toast” in Android is a popup message that sticks around, then vanishes.

User Interaction: Example 8

- Let’s break it down into concrete tasks.
- We’ll have to create the UI. Assume that’s done.
- We’ll have to detect the click of a submit button.
  - We’ll have to read the text in the EditText, and store it in a Kotlin string.
  - We’ll then have to *parse* that string (we’ll use regex).
    - Get the first and last name, assuming space is a separator.
    - Handle all edge cases here (null, blank, etc).
  - Then we’ll display a short-lived message for each case.

User Interaction: Example 8 (button click)

- Let’s focus on the button click.
- What we need is a *callback*: a function that is automatically called/invoked when an event occurs.
- Fortunately, Android has a bunch of callbacks already in place for several standard types of user interaction.
- There are many ways to handle button clicks built into Android.
- Most of them involve implementing the View.OnClickListener class in some way.

User Interaction: Example 8 (button click)

- Here's one simple way:

```
button.setOnClickListener(object : View.OnClickListener {
    override fun onClick(view: View?) {
        // Do some work here
    }

})
```

- Here's the same thing using a lambda instead:

```
button.setOnClickListener(View.OnClickListener { view ->
    // Do some work here
})
```

- The problem is that we'd have to do this in-place for each button.

User Interaction: Example 8 (button click)

- This resutls in wasteful and error-prone code.
- There’s a related but better way to do this.
- Have your Activity class implement View.OnClickListener.
- This is an interface.
- If you implement this, you **must** override the onClick() method in this interface.
- All the click functionality is handled in the overridden onClick() method.

User Interaction: Example 8 (string parsing, messages)

- In the onClick() method, we need to:
  - Grab the string from the EditText.
  - Parse it to figure out first name and last name.
  - Set first name to a TextView, and last name to another TextView.
  - Show a message to the user for each edge case.
- How do we show pop-up messages that vanish?
- These are called Toasts in Android.
- Let’s see a quick example.

User Interaction: Example 8 (Toasts)

- Here's how you create a Toast that displays a string message:

```
Toast clickToast = Toast.makeText(this@MainActivity, "Good job!",
Toast.LENGTH_SHORT).show()
```

- Or, if you'll never need the toast object again:

```
Toast.makeText(this@MainActivity, "Good job!",
Toast.LENGTH_SHORT).show()
```

- Okay, let's put it all together in Example 8.

User Interaction: Example 8 (summary)

- You learned (or revisited) button-click handling.
- You did some event-driven programming, moving text around between different Views (widgets).
- You did some behind-the-scenes edge case handling and string parsing.
- Let’s move on to Intents.
- I know you’ve done these. Recall what they’re used for...

User Interaction: Intents

- All Android activities are started or activated with an Intent.
- You also use Intents to communicate between Activities.
- Activity A can pass data along with the Intent, and Activity B can receive it and do something with it.
- There are really 2 types of intents: explicit intents, and implicit intents.
- Explicit intents explicitly start a specific Activity in your app.
- Implicit intents simply allow the OS to decide which Activity should be started.
- For instance, you may want to pass a URL to Maps using an implicit intent, or start the camera app to take a picture.

User Interaction: Intents

- Intents contain the following:
  - An *optional* **component name** (the one you want to start). This decides whether intent is explicit or implicit.
  - An **action**: for instance, whether you want to view stuff, or share stuff.
  - **Data**: This has to be a URI object that references the data to be acted on, and/or the type of data.
  - Other stuff (strings, data, etc.).

User Interaction: Explicit Intents

- Let's say we're starting a new Activity called ActivityB from the MainActivity.
- Let's also say we want to send some data over from MainActivity to ActivityB.
- The most general way is to send a **Bundle** object.

```
val messageIntent = Intent(this,
ActivityB::class.java)
val messageBundle = Bundle()
messageBundle.putString(KEY,value)
messageBundle.putInt(KEY,value)
messageIntent.putExtras(messageBundle)
startActivity(messageIntent)
```

- Let's do a quick example based on Example 8.

User Interaction: Example 9

- Remember Example 8?
- We'll move the TextViews over to a second Activity.
- We'll send the EditText string to the second Activity upon a button press.
- We'll then break up the string in the second activity, and display the first name and last name.
- Not complicated, but demonstrates explicit intents nicely.
- We can add a back button to the action bar by including the child activity in AndroidManifest.xml.
- Make the program safer? (Exercise for you).

User Interaction: Example 10 (Maps)

- Implicit intents allow the OS to decide which Activity should be started rather than explicity specifying it.
- Let’s do a Maps example.
- We can use an implicit intent to launch the map.
- WEB has lat:long 40.767778,-111.845205.
- Let’s write an app to search for stuff near these coordinates.
- It’s a bit contrived, but useful to know.

User Interaction: Example 10 (Maps)

- This time, one Activity will suffice.
- Handle the button click by:
  - Grabbing the search string.
  - Constructing a URI (Uniform Resource Indicator).
  - Passing the URI to an implicit intent with the ACTION_VIEW attribute, which says we want to view it.
  - Handle the possibility that this intent may not be bound to any particular package/app/activity.
  - Start the appropriate activity!

Digression: URI

- A URI is a string of characters.
- Meant to unambiguously identify resources.
- A URL (Uniform Resource Locator) is just a specific type of URI that tells you both where to find a resource and how to act on it.
- Eg: http://www.cs.utah.edu/~benjones tells you to get the resource “~benjones” using http: (hypertext transfer protocol) from the network host with domain name www.cs.utah.edu.
- Generic URI syntax:

  URI = scheme:[//authority]path[?query][#fragment].

- We’ll discuss this more when we link our app to the web.

User Interaction: More implicit intents

- The goal isn’t to show you all possible implicit intents.
- I do want to show you the fun/useful ones.
- Each implicit intent is a lesson! We learned a little bit about URIs, now we’ll learn about media types.
- Think of implicit intents as avoiding reinventing the wheel.
- If there’s app for it, and it’s good, use it.
- Another good example along these lines is the camera.
- If you want to take a picture, just use the camera intent.

User Interaction: Example 11 (Camera)

- Imagine you’re building a user profile page for your app.
- You take in the full name, split it into first and last names.
- You may also want to take a profile pic, and store it.
- Let’s modify Example 8.

User Interaction: Example 11 (Camera)

- We'll just add a "take profile pic" button to our earlier UI.
- When you click it, it lets you take a pic using the camera API.
- You get to use the camera's controls to do it.
- Then, you return a little thumbnail to store on the app.
- Later in the course, we'll use this to work with full images and put in facial recognition into your app.
- Implicit intents make this whole thing pretty straightforward.

User Interaction: Example 11 (Camera)

- First, we add a Button and an ImageView.
- Clicking the Button should take you to the camera.
- Once you click a picture, a thumbnail is returned.
- The ImageView is populated by the thumbnail image.
- What are the coding steps to do this?

User Interaction: Example 11 (Camera)

- We know most of this stuff already.
- We already have a Button click handler from Example 8.
- We'll just add another "case" to the switch statement there.
- When the button is clicked, we start a new implicit intent.
- The implicit intent must be passed a parameter of MediaStore.ACTION_IMAGE_CAPTURE.
- However, this time, we expect the implicit intent to actually return data.
- We need a callback to handle the return of control from the camera.

User Interaction: Example 11 (Camera)

- Until API 28, we could have just called startActivityForResult(). This is now deprecated.
- Now, we'll have to register a callback that can get triggered when the camera activity returns.

```
private val cameraActivity =
registerForActivityResult(ActivityResultContracts.StartActivityForResult()){result->
    if(result.resultCode == RESULT_OK) { // Do stuff here}

}
```

- The result object is of type ActivityResult; it contains a *resultCode* and a *data* field. The latter is an intent containing a bundle in its *extras* field.
- We grab the data from the extras field, cast to a Bitmap object (since we know it returned a Bitmap).
- We then populate the ImageView with that Bitmap object.

User Interaction: Summary

- We learned about ConstraintLayout, and a few important views.
- We learned to handle click events.
- We revised explicit intents, and message passing between activities in your own app.
- We reviewed some important implicit intents: maps, camera.
- We learned how to handle data returned from an implicit intent.

Activities: Lab

- Let’s do a lab where you implement some of these views.
- Implement the UI for the sketch on the board using both ConstraintLayout.
- Only use design mode for ConstraintLayout, and be careful to check text mode occasionally.
- Implement the functionality as described on the board.