Lecture 2: Activities and Intents

Recap

- We saw the basics of sticking Views on the screen (Activity with layouts)
- We saw simple interaction (button's click listener)
- We saw a little bit of Kotlin

Today

- Neat trick: “databinding”
- User-facing-debugging: Toasts
- Linking Activities: intents
- Implicit intents: Maps and Camera

Data binding

- We saw yesterday how to use: `val myView = findViewById<viewtype> (R.id.whatever)`
- That's kind of annoying, and somewhat error prone (what if you get the ID wrong?)
- Data binding is a nice improvement to that
- Like the R class that is automatically created based on the XML, another class is created which has a property (member variable) for each View (that has an ID)
- With that you can just do `binding.someID` to get a reference to the view, with the correct Kotlin type

Data binding details

- In the “app” gradle file (there's also a “module” one, don't edit that one) add this to the android section:

```
buildFeatures {
    viewBinding = true
    dataBinding = true
}
```

- For each Activity/Fragment XML file, the build system will create a class with the Camel-Cased'd name with “Binding” at the end
- In your activity's `OnCreate` method, you'll inflate the binding and can then use it

Data binding code example

```
val binding =
ActivityMainBinding.inflate(layoutInflater)
binding.button.setOnClickListener{ }
setContentView(binding.root) //use this setContentView overload instead of the ID one
```

Discussion: Is this good?

Toasts

- A toast is a little popup message that disappears after a couple of seconds
- They're a quick and dirty way to inform a user that something happened
- We'll use them in our examples to tell users about invalid inputs (among other things)
- Like other things we'll cover, they're probably not a good idea to include in a production grade app

Toast details

```
val myToast = Toast.makeText(this, "message", Toast.LENGTH_SHORT)
myToast.show()
```

- This can be a one-liner if you want to fire and forget it (and don't want to keep a reference to the toast)

Example: Email splitter with toast

Intents

- Activities are started with an Intent
- Intents can contain data that is passed from one Activity to another
- With “Explicit Intents,” we specify the Activity we want to start
- With “Implicit Intents” we say what we want to do and let the OS pick which activity should do it (If we want to “Share” something the user can pick between Messages, Gmail, Twiter, Slack, etc)
- Note: as I said, modern apps don't use explicit intents, since they only have 1 activity

Intent Details

Intents contain:

- A “Component name” if it's explicit
- An “Action” for implicit intents
- Data to be sent to the new activity/app

Example: Explicit Intent

- We want to start `ActivityB` after some interaction in `MainActivity`
- And we want to include some data
- We put the data into a “Bundle” (which is pretty much `HashMap<string, object="">`)

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

Getting the data in the new activity:

```
//implicitly calls getters
val theExtrasBundle = intent.extras!!
val someVal = theExtrasBundle.getString("someKey")
```

Example: email splitter

Implicit Intents

- For implicit intents, we let the OS/user pick which app opens when we create the intent
- We'll see how to use an intent to start the Maps app (or another mapping app if the user wants)
- We'll pass the GPS coords of WEB via the intent and try to search for stuff nearby

Map search plan

- We'll have one activity
- When the user clicks the button we'll:
  - Grab the search string from an EditText view
  - Create a `URI` (Uniform Resource Indicator) to pass to the Intent
  - Pass it to the intent using the `ACTION_VIEW` attribute which says we want to “View” that URI
  - Handle errors if there are no apps that can open our URI

URIs

- URIs are strings that identify resources
- URLs are a subset of URIs and probably we can use the 2 terms interchangeably (though they are not always interchangeable)
- A URL format is `<scheme>:[//authority]<path>[?query][#fragment]`
- Common schemes are `http`, `https`, `ssh`
- `https://msd.utah.edu/#jn-staff` is a URL and URI that refers to a resource you can access via HTTPS by contacting msd.utah.edu
- To open a maps app, we can use the “geo” scheme

Example Search in maps

Another example: The camera

- There's a bunch of implicit intents which we'll see throughout the course
- If users can use an app they like to do something, and you don't have to write code to put it in your app, GOOD! Don't reinvent the wheel
- Don't deal with low level camera APIs if you don't have to; open the Camera app and use the picture it gives you back

Example app: Camera

- This might be useful if there's a “User Profile” part of your app, and you want a profile pic
- We'll add this into an app which takes the user name and splits it into first/last
- Users can take a profile pic, store it, and can see it displayed in the app

Details

- We'll add a button to our UI to open the camera
- We'll open the Camera app with an implicit intent
- We'll add a callback to react when the camera gives us a picture back
- Then we can set an ImageView's content to be the thumbnail we got
- We'll see later how to get the full size image (this intent mechanism doesn't do that)
- Note: some emulated phone cameras don't work all the time... sweet

More details

- Since we made our Activity implement the click handler interface, we can pass it as the handler for both buttons
- Inside we can look at the ID of the button that was clicked to decide which action to take
- Our intent action is `MediaStore.ACTION_IMAGE_CAPTURE`
- And we need a callback for what to do when the camera gives us our thumbnail

Grabbing the Thumbnail

- To get the thumbnail we register that we want to start and activity, and that we expect that activity to return a result to us
- As part of this request we specify a callback to be executed when we get the result back from the activity (from the camera in this case)
- We store info about this request as a member variable so that it has the same lifetime as the activity
- In it, we'll update the ImageView's contents

```
private val cameraActivity =
  registerForActivityResult(ActivityResultContracts.StartActivityForResult()){result->
  if(result.resultCode = RESULT_OK) { /* called when we get a thumbnail back */ } }
```

Kotlin Syntax Note

- It's common for functions to functions as parameters (for callbacks, or for “functional” operations like `map` or `filter`)
- There's a special shortcut syntax for that in Kotlin (and also Swift):

```
funcThatTakesAFunction(a, b, c,
    {x, y -> x + y})
```

- The last parameter is a function taking x and y and returning their sum
- It's hard to read the closing `}` and `)` sometimes, so if the last parameter is a function, you can just put the whole function after the closing parens:

Trailing Lambda Syntax

```
funcThatTakesAFunction(a, b, c){
  x, y -> x + y
}
```

- This does the same thing as the previous slide
- Another example:

```
val numbers = setOf(1, 2, 3)
println(numbers.map { x -> x * 3 })
//prints 3, 6, 9
```

Back to the Camera Example

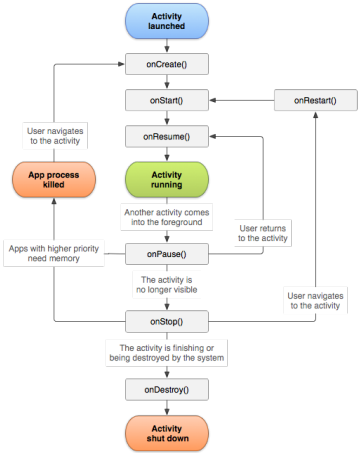
- The parameter passed to our “activity restult callback” is an “ActivityResult”
- It contains a `Bundle` whose `data` member has our `Bitmap` in it

Camera Example

Activity Lifecycle

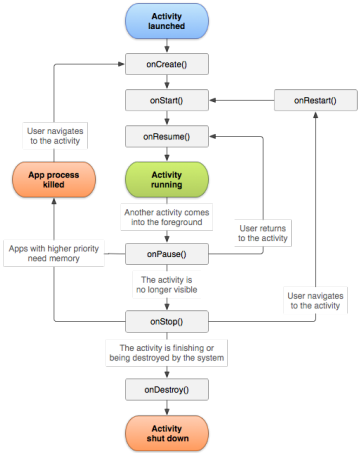
- The OS forces the app to change state through user interactions
  - user switches to a different app
  - phone needs to free the memory your app is using
  - user rotates their screen
- Without extra care, we will lose data when these changes happen
- We'll learn about writing “lifecycle aware” apps

Activity Lifecycle



Activity Lifecycle

Activity Lifecycle



Activity Lifecycle

Activity Lifecycle

- Activities transition between a set of states through user/OS interactions
- You can add callbacks which the OS will call when those transitions occur
- For example, you may want to save data before calling an Intent, and restore it if the user comes back to that activity
- Overriding `onPause()` and `onResume()` is one way to do that

Lifecycle example

- You open Google Play, download an app to install
- A screen appears, asking you about app permissions
- The activity for the install screen is paused, permission screen is visible
- However, the install screen is also still visible!
- So, the UI still needs to be drawn for the install screen activity. However, the rest of it needs to be paused
- When you think of the activity lifecycle, you are forced to think of how different parts of your activities need to be handled differently

Email splitter with screen rotation

- In the single activity “email splitter” happened when the screen was rotated?
- The text views get cleared...
- When the screen is rotated, the activity is `destroyed` and then `created`
- Our callback that fills in those views only runs on button clicks, so the newly created views have the default values
- The `EditText` DOES remember what the user typed

Storing data persistently

- We need our data, which we want to show in the `TextViews` and the `EditText` to be saved across Destroy/Create state changes
- We saw that the system will call some callbacks defined in our activity on state changes
- Which of those are most helpful here? Actually there'a few options
- How do we actually save the data in those callbacks?

Solution for small amounts of data

- For small amounts of data we can store what we need during the `onSaveInstanceState()` method, which will be called before the activity is destroyed
- We can grab the data and update our views in `onRestoreInstanceState()`, or in `onCreate()`
- The Save/Restore state methods are called before/after `onStart`/`onStop`
- Those methods take a `savedInstanceState` parameter which is a `Bundle` where we can store/retreive small amounts of data.

Code example

```
override fun onSaveInstanceState(savedInstanceState: Bundle?) {
  // Always call the superclass so it can save any view hierarchy
  super.onSaveInstanceState(savedInstanceState)
  // Save the user's current state
  savedInstanceState.putInt("myInt",100)
  savedInstanceState.putString("myString", “value”)
}

override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
  mTv!!.text = savedInstanceState.getString("myString")
}
```

Example doing the restoration in OnCreate

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState) //call the superclass first
  if(savedInstanceState!=null){
    mSomeInt = savedInstanceState.getInt(KEY);
    mSomeString = savedInstanceState.getString(KEY);
  } else {
    //Initialize stuff as you would've for the first time
  }
}
```

Fixing the email splitter

- We could store the contents of the TextView objects before they're destroyed, and fill them in when they're (re)created
- We could also “fix” this problem by disabling screen rotation
- That decision should be based on whether it makes sense to use the app in both portrait/landscape mode or not. It should not depend on whether you want to fuss with making the app lifecycle aware
- The restoration can happen in either `onCreate` or `onRestoreInstanceState`
- There are subtle differences between these two approaches, for example, `onCreate` is not called when the app “restarts” (the user navigates to another activity and back)

Gut check: Is this good?