Fragments and Custom Views

Recap

- Screens can be implemented with Activities, data passed via intents
- ViewModels are nicer than passing bundles, so can just use those!
- ViewModels are usually associated with one activity, so how do we share across screens?
- One approach, use a single activity, many Fragments
- There's some other benefits to this approach

Recap Fragments

- A fragment is a reuseable piece of the UI
- It could be something small like a “contact card” with a picture + some text
- It could be a whole screen

Example use of Fragments: Master/Detail view

- We have a list (recyclerview) of items. When we click it, we see the details about a single item in a larger view
- On a phone, the details might take up the whole screen
- On a tablet, it might be shown next to the list
- If the “list” part and the “details” part are both created as Fragments, it's easy to do the layout on either device while reusing most of our code

Creating a Fragment

- Basically the same as creating a new activity. Right click in the project → new → fragment (there's a few reasonable options for fragment templates there)
- Like an Activity, you get an xml layout and a kotlin file with some of the lifecycle callbacks filled in for you
- Fragment lifecycle callbacks are similar to, but not quite the same as the Activity ones

Static Fragment in an Activity

- If you want an activity to always display the same fragment, you can add the fragment directly to the Activity's layout:

```
< SomeLayout ... >
< fragment
    android:name="com.example.Fragment1"
    android:id="@+id/Fragment1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>
< /SomeLayout>
```

- The android:name attribute is the Fragment class you're adding

Embedding Fragments Dynamically

- You can also embed Fragments dynamically.
- For example: you may want to switch between 2 Fragment at runtime, based on some user input
- You need to write both XML and Kotlin code here
- First, in the xml file, put a placeholder FrameLayout or FragmentContainerView where you know you'll insert the Fragment
- Then, in the Kotlin file, use supportFragmentManager to create a FragmentTransaction object
- FragmentTransaction allows us to add fragments to the FrameLayout atruntime, but this is processed as a transaction (all or nothing, remember 6016?)

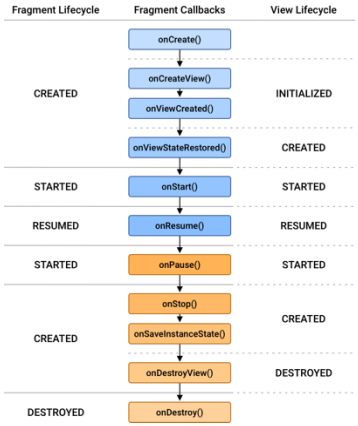
Fragment creation example

```
val myFragment = MyFragment()
//Send data to it, accessible in the Fragment as "arguments" property
val sentData = Bundle()
sentData.putString("SomeKey", "SomeValue")
myFragment.arguments = sentData
//Replace the fragment container
val fTrans = supportFragmentManager.beginTransaction()
fTrans.replace(R.id.fl_frag_container, myFragment, "some_tag") fTrans.commit()
```

Accessing Fragments from Code

- If you embed a Fragment statically, you can just use `findViewById()` or binding like any other view
- But if you don't embed it statically, there's no ID in the XML file, so you can't look it up that way.
- An alternative is to add a “Tag” and use `View.findViewWithTag()`
- Or, you can track the `Fragment` object from Kotlin and store it in a variable

Fragment Lifecycle



Fragment Lifecycle

Important Callbacks

- onCreate(): called when created. Views don't exist yet, so what you can do is limited
- onCreateView(): XML layout is inflated. This is usually the only one I implement
- onViewCreated(): called after onCreateView() and should contain other initialization logic
- onPause()
  - called when user navigates backward, or the fragment is replaced/removed, or if the activity is paused
  - A good place to save data to persistent storage

Custom Views

Custom Views

- You can make your own View classes by writing a class that inherits from `View`
- Your constructor will need to look like this (you don't call the constructor in your code, so you can't change the signature): class `myView(context: Context, attrs: AttributeSet) : View(context, attrs) {`
- There's several methods you can override, but you probably want to override `onDraw(canvas: Canvas)` at minimum
- As a general rule, your view should be “dumb” and not contain more than you need to do the drawing
- If you need more “smarts” ie to access a viewmodel, define a Fragment that contains one of these views

Canvas

- Your onDraw takes a `Canvas` ... what's that?
- A Canvas is an object that has methods for drawing (`drawRect` etc)
- Most drawing commands take a `Paint` object which is basically the "pen" we're drawing with
- When you create a canvas you pass the thing you'll be drawing on, which could be, for example a `Bitmap` image
- Your onDraw receives a Canvas which is "connected" to the pixels of your view, so whatever you draw on that canvas will show up on the view

Bitmaps

- A `Bitmap` represents an image (an image made of pixels, not a vector image)
- The `Canvas` class has as `drawBitmap` method that copies the bitmap (possible with scaling or other transforms) to the canvas
- You can also create a Canvas which is wrapped around a Bitmap, so any drawing commands show up in that Bitmap

Custom Drawing View

- This is sort of roundabout but here goes
- Create a custom `View` class
- In it's `onDraw(Canvas)` method, we want to call `drawBitmap` with a `Bitmap` containing our custom drawing
- It's probably best to create that bitmap in a class that has a longer lifecycle, but store a reference in the custom view class
- To draw to that bitmap, we'll create a `Canvas` that wraps it, and then we can run drawing methods on that Canvas
- So we'll be dealing with 2 canvases: One which lets us draw on a bitmap, and one which draws out bitmap to the screen