MVVM Architecture

Also RecyclerView

Recap

- Activities (and views/fragments) undergo “lifecycle changes”
- Example: when the screen is rotated, the running activity is destroyed and a new activity is created
- For data that needs to persist across lifecycle changes, we can store/retreive it in a bundle
- onSaveInstanceState and onRestoreInstanceState are called as appropriate so we can do so before destruction/at creation
- This is manual and error prone!

MVVM Architecture

- Quick detour to high level description before we see android details (which solve the lifecycle problem)
- MVVM pattern has us split our app into 3 pieces:
  - Model — the App data
  - Views — What the user sees, generally “dumb”
  - View Model — Tracks changes to model data and informs the views of updates. Views and tell the viewModel about user interactions

Naive architecture

- Imagine a “to do” list app where there's a text entry + button to type new tasks, and list showing all the added items
- The “naive” implementation of the app would have the button grab the text of the new item, append to the list, then have the list be redrawn
- This works for an app with simple relationsips between views
- BUT!!
- This requires the button's click handler to know about all the places in the app that use the new data
- Maybe the list of items is stored in a List as well as in a view displaying the list. The button handler must update both

MVVM version

- When the button is clicked, it's click handler calls a VM method “addNewItem(String)" or similar
- The VM stores List with the “single source of truth” (SSOT) about what tasks there are, which is updated
- Any view that depends on that list (the list view, and maybe another view which shows the number of todo items) **observes** that list and will be notified when it changes
- In a good MVVM implementation, the observers are notified automatically

MVVM Advantages

- Views that need to update model data no longer care who's interested in that data. They simply tell the viewmodel “please change the model”
- View's dependencies are made explicit, and written in one place in the code. They register themselves as observers for data provided by the viewmodel
- This separates/simplifies the updaters and the updatees

MVVM in Android

- View models are defined in classes that inherit from ViewModel, which handles “lifecycle awarness”
- Data in the viewmodel class will persist across lifecycle changes for activities, etc
- Your VM class should have methods that views will all to updateapplication data (addTask or markTaskCompleted) etc
- Data that your views want to observe should be provided by LiveData “properties”

LiveData

- LiveData is a generic type (observableName: LiveData<string>) which stores a value.
- You can call observe on the livedata and pass a callback which will be run whenever the value changes (and when it's initialized)
- Rarely (almost never in your views!) you can access the current value with the value property, but the more common use of value is for your viewmodel to update the livedata

Kotlin properties

- Code using an object can access a property with object.propertyName to either read from it or write to it
- It could be implemented in the class just like a public member variable (public var x : Int)
- It could also be implemented with a getter and a setter, but the caller doesn't need to know that!

```
private var _backingVariable var
myProperty: Int
    get() = _backingVariable*2
    set(newValue) { _backingVariable = newValue/2 }
```

- Read only properties are declared with val, details here

Kotlin delegated properties

- A “delegated” property is a property that calls another object's get/set as appropriate
- We'll see it used in quite a few places
- `val myProperty by lazy { expensiveInitialization() }`
- `lazy` is a type which takes an initialization function. It runs it the first time we access the property and stores the result
- So we get a value who's expensive initialization is performed on first use, and then remembered after
- `lazy` doesnt' have a `set` method, so it's for read only properties

Recyclerview

- Pure nightmare fuel, just awful!
- Solves a difficult problem with some tough constraints
- Special view for displaying a list of items in a memory efficient manner
- Reuses view objects for items the scroll off the screen to keep memory usage low
- Supports reasonable choices of customization (leading to a pretty complex system!)

Recycler View pieces

- We need to provide a few different things to the RV:
  - A layout manager (how the list items should be laid out)
  - A view holder (the “View template” for list items)
  - An Adapter (responsible for creating Views when needed)
  - An Animator (responsible for animating changes to the list)

Layout Manager

- The layout manager controls how the list items are laid out
- You can pick from this list:
  - LinearLayoutManager — basically like the LinearLayout class we've seen
  - GridLayoutManager — I bet you can guess
  - StaggeredGridLayoutManager — “Pinterest style” offset grid
  - Implement your own class that inherits from RecyclerView.LayoutManager

View Holder

- This class describes how to turn a list item into a view
- These classes inherit from RecyclerView.ViewHolder
- You specify the View or ViewGroup template for a list item
- The RecyclerView automatically rebinds the data for the visible items so that the actual number of views that are created/destroyed is small
- For an “infinite scroll app” this means you'll only have about as many views as fit on the screen at a time (plus a small buffer)

Adapter

- The adapter is the object that manages creating/destorying/rebinding the ViewHolder objects
- Adapters inherit from `RecyclerView.Adapter`
- You'll override a few methods which are called when the view is updated:
  - `onCreateViewHolder` — when a new ViewHolder is created
  - `onBindViewHolder` — when a ViewHolder is recycled
  - `getItemCount` — tell the RV how many items there are

Animator

- Animators alter the appearance of views in the list
- As with most RV stuff, you inherit from a nested RV class: `RecyclerView.ItemAnimator`
- `DefaultItemAnimator` gives us a pretty good starting point, and is used by default (duh)

Example: Todo app

- The todo list itself store in a ViewModel class (just a list of objects, with some wrapping). In this case the ViewModel is also handling the Model (we'll split this responsibility in the future)
- The list is exposed to the views as LiveData<list<listitem»
- The VM has methods that the view can call to update the data (add to/remove from the list)
- In Activity.onCreate we register an observer to the LiveData in the viewmodel which will run whenever there are updates

Example: RecyclerView

- The activity has a `RecyclerView` to display the list items
- Associated with it are:
  - A Layout XML file describing what each todo item should look like
  - A data class representing a single item
  - An Adapter class with a nested ViewHolder class
- Interesting thing: the Activity is responsible for passing a callback to run when we click buttons on the todo items... good idea?
- Easiest to understand the pieces by looking at the code

Discussion

What did we do well/poorly in this design?