ATLS 4320: Advanced Mobile Application Development
Week 12: ViewModel and LiveData

Until recently, Google did not recommend a specific approach to building Android apps. They just provided the SDK and tools and developers could decide what architecture they wanted to use for their apps. This resulted in a wide range of approaches, many resulting in "God activities" – where all the code was put into one large activity class. This resulted in large, complex classes that were hard to maintain.
In 2017 Google introduced the Android Architecture Components which became part of Android Jetpack when it was released in 2018. These components provide the building pieces needed to make it easier to create components that support single responsibility components with separation of concern. These components are designed to make your app better, more robust, and easier to maintain, but there is a cost, both in learning curve and in initial development time.
https://developer.android.com/jetpack/docs/guide
Although not forced to use them, this is clearly the path Google is taking moving forward. One of the most important rules for architecting your application is to figure out for yourself for your particular app which of these components are going to make your job easier both for initial development and long-term maintenance.
We're going to look at two of these components today and see how we can use them in our app development.

ViewModel
https://developer.android.com/topic/libraries/architecture/viewmodel
ViewModel is an architecture component designed to store and manage view-related data in a lifecycle conscious way. Using a ViewModel as the data manager separates ownership of the data and logic from Activities or Fragments. Activities and Fragments should represent the visual user interface and only have code that affects the user interface presentation. All of your business logic and any code that goes to managing data in memory should be stored by the ViewModel, therefore supporting single responsibility for our components. Each activity or fragment has its own ViewModel and the ViewModel that persists between configuration changes including screen rotations so you don't have to deal with life cycle events as much.

- Regardless of how many times a UI controller is recreated during the lifecycle of an app, the ViewModel instances remain in memory thereby maintaining data consistency.
- A ViewModel used by an activity will remain in memory until the activity completely finishes (activity is destroyed or in a single activity app, the app exits).
- After it has been recreated due to a screen rotation or other event, for example, it is still the responsibility of the UI controller to re-populate the user interface with data from the ViewModel.
- It is also the responsibility of the UI controller to identify when data has changed within the ViewModel so that the user interface can be updated accordingly.
- Since the ViewModel will outlive the activity/fragment it's associated with it shouldn't contain references to any UI controllers or views.
    - The one exception is if the model needs access to the Application context. This is ok because an Application context is tied to the Application lifecycle which a ViewModel won't outlive. (This is different from an Activity context, which is tied to the Activity lifecycle.)
    - If you need the Application context use the AndroidViewModel class instead as it's basically a ViewModel that includes an Application reference.

- ViewModels handle transient data during the app's lifecycle, but they don't handle data persistence between app launches.

To use the ViewModel component you will need to add the dependencies to the app's grade file:
```
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1"
```

To create a ViewModel class you create a class and extend the ViewModel class.
```
class ItemViewModel: ViewModel() { … }
```

To create an instance of your ViewModel in an activity or fragment you can use *by viewModels(). by viewModels()* is a Kotlin extension function that gets a viewModel with a lifecycle tied to this activity. This is new in Android ktx module version 1.1.0 so you don't need to define a ViewModelFactory class anymore.

```
val viewModel: ItemViewModel by viewModels()
```

(note that you will see the ViewModelProviders.of() method used in many online tutorials. This class has been deprecated and should no longer be used.)

You can create an instance of a ViewModel class in multiple activities/fragments and each will be specific to that activity/fragment and store different data.
SaveInstanceState, ViewModel, and local data persistence can all work together, depending on implementation needs.

- Local data persistence is used for storing all data you don't want to lose if you open and close the activity.
    - Example: The collection of all song objects, which could include audio files and metadata
- ViewModels are used for storing all the data needed to display the associated UI Controller.
    - Example: The results of the most recent search, the most recent search query
- onSaveInstanceState is used for storing a small amount of data needed to easily reload activity state if the UI Controller is stopped and recreated by the system, not for complex data. Instead of storing complex objects here, persist the complex objects in local storage and store a unique ID for these objects in onSaveInstanceState().
    - Example: The most recent search query

LiveData
https://developer.android.com/topic/libraries/architecture/livedata
LiveData is an observeable data holder class. It is a wrapper around your view models that notifies observers when data in that model changes. LiveData ensures that the user interface always matches the data within the ViewModel. This eliminates the need of the controller to continuously check with the ViewModel to find out if the data has changed since it was last displayed.
LiveData follows the observer pattern. Instead of updating the UI every time the app data changes, an observer can update the UI every time there's a change. This allows you to consolidate your code to update the UI in these Observer objects.

- The LiveData object can be subscribed to from the presentation layer. Whenever there are changes in the data, those changes are published and then the presentation layer reacts.
- A LiveData instance may also be declared as being mutable, allowing the observing entity to update the underlying value held within the LiveData object. The user might, for example, enter a value in the user interface that needs to overwrite the value stored in the ViewModel.
- LiveData is an abstract class, MutableLiveData is an extension of that class.

LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. LiveData objects know when the activity's lifecycle state changes and respond accordingly. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

- If the activity is paused (perhaps the app is put into the background), the LiveData object will stop sending events to the observer.
- If the activity has just started or resumes after being paused, the LiveData object will send a LiveData event to the observer so that the activity has the most up to date value.
- LiveData instances will know when the activity is destroyed and remove the observer to free up resources.

To use the LiveData component you will need to add the dependencies to the app's grade file:
```
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.3.1"
```

LiveData is a wrapper that can be used with any data, including objects that implement Collections, such as List. A LiveData object is usually stored within a ViewModel object.

You then create an Observer object that defines what happens when the data in the LiveData object changes. The Observer object is created in a UI controller (activity or fragment) and usually updates the UI to reflect the change in data.

Then attach the Observer object to the LiveData object that you want to observe using the observe() method. Every time that data is updated, the observer will be notified, the onChanged method for the observer will be called, and the UI updated.

LiveData delivers updates to active observers when data changes. An exception to this behavior is that observers also receive an update when they change from an inactive to an active state. Furthermore, if the observer changes from inactive to active a second time, it only receives an update if the value has changed since the last time it became active.

LiveData has no publicly available methods to update the stored data. Usually MutableLiveData is used in the ViewModel and then the ViewModel only exposes immutable LiveData objects to the observers. The MutableLiveData class exposes the setValue() and postValue() methods publicly and you must use these if you need to edit the value stored in a LiveData object. Both of these methods result in their observers to call their onChanged() method. In Kotlin both are called through .value.

List
Create a new project
Basic Activity template
Name: List
Package name: the fully qualified name for the project
Language: Kotlin
Minimum SDK: API 21 (21 is the minimum API for Material Design)
Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)
Finish

Add the dependencies for the viewmodel and livedata components into the app's grade file
```
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1"
```

```
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.3.1"
```

Add colors, create a theme with a Material Components parent theme, and apply it to your app.

The basic template includes a little more than the empty template.
activity_main.xml sets the AppBarLayout which includes a Toolbar.
Includes the content_main xml layout which is a fragment that is the navigation host fragment.
Our app won't need navigation so we'll just use content_main for our layout.
Includes a floating action button.

MainActivity.kt sets up the floating action button as well as an onClickListener for it.
A menu has also been added and set up.

List Layout
content_main.xml
Remove the fragment
In design view drag out a RecyclerView to display the list (can be found in common or containers).
Add start, end, top, and bottom constraints.
Make sure the RecyclerView has layout_width and layout_height set to "match_constraint" (0dp).
Give the RecyclerView the id recyclerView.

List Item Layout
We also need a layout file to determine what each row in our list will look like. Ours will be super simple with just a TextView.
File | New | Layout resource file
File name: list_item
Root element: androidx.constraintlayout.widget.ConstraintLayout
Source set: main
Directory name: layout

Drag out a TextView and make sure it has an id (textView).
Make sure the TextView has layout_width and layout_height set to "wrap_content".
Add missing constraints.
I also added some bottom padding so the text is in the vertical center of the row and made the text larger.

```
android:textAppearance="@style/TextAppearance.MaterialComponents.Headline6"
android:paddingBottom="10dp"
```

Once you don't need the default text to help with layout, remove it.

Also change the constraint layout's height to "wrap_content". If the height is match_constraint each text view will have the height of a whole screen.

Kotlin class
We're going to create a custom Kotlin class to represent the data we'll be using.
First create a new package for our model. In the java folder select the list folder (not androidTest or test)
File | New | Package
Name: model

Then select the new model package and add a new class called Item.

File | New | Kotlin class
Name: Item
Kind: Class

The Item class will be a data class that will store the item name that will be defined in the primary constructor in the class header.

```
data class Item(val name: String){}
```

ViewModel
Now we'll create our view model class.

Select the model package and add a new class called ItemViewModel.
File | New | Kotlin class
Name: ItemViewModel
Kind: Class

Our ItemViewModel class should subclass the ViewModel class.
We'll also add a MutableLiveData List that we'll later observe in MainActivity.
We'll also use a 'reference list' in our ViewModel so we can operate on it and use it to update the LiveData with this list. Since Kotlin is a pass-by-reference language, the value that MutableLiveData is holding is just a reference to the memory where the array list is held, which means that if you change the contents of the array, the MutableLiveData.value has not actually changed. Using this reference list, newList, we can set a new value on the MutableLiveData array list in order to update it and notify the observer.

```
class ItemViewModel: ViewModel() {
    val itemList = MutableLiveData<ArrayList<Item>>()
    private var newList = ArrayList<Item>()

    fun add(item: Item){
        newList.add(item)
        itemList.value = newList
    }

    fun delete(item: Item){
        newList.remove(item)
        itemList.value = newList
    }
}
```

Adapter
Now we need to create an adapter which will be used to populate the data into the RecyclerView. The adapter's role is to convert an object at a given position into a list row item to be inserted in the list.

In the java folder create a new class for our adapter.
File | New | Kotlin Class
Name: MyListAdapter
Kind: Class

The class will need to subclass RecyclerView.Adapter

```
class MyListAdapter: RecyclerView.Adapter {}
```

AS should add the import for the RecyclerView class:

```
import androidx.recyclerview.widget.RecyclerView
```

With a RecyclerView the adapter requires the existence of a ViewHolder object which describes and provides access to all the views within each item row.
To indicate that our class will be implementing the ViewHolder class add it to the class definition.

```
class MyListAdapter: RecyclerView.Adapter<ViewHolder>() {}
```

You will give you errors until you implement the required methods.
Select the light bulb and choose implement methods to get rid of the error.
Every adapter has three required methods that we need to implement.

You will still have an error because we need to define the ViewHolder class which will set up the bindings to the view in the layout file.
In our MyListAdapter class we'll create a ViewHolder class to render the item and a default constructor.

```
class ViewHolder(view: View): RecyclerView.ViewHolder(view) {
    val itemTextView: TextView = view.findViewById(R.id.textView)
}
```

This should have fixed the errors.
With the viewholder defined, let's set up the MyListAdapter class. We'll define a view model and add it to the primary constructor.

```
class MyListAdapter(private val itemViewModel: ItemViewModel) :
RecyclerView.Adapter<MyListAdapter.ViewHolder>() {}
```

Create an array with the items from our view model at the class level.

```
private var myItemList = itemViewModel.itemList.value
```

Now we'll implement the 3 required methods.

OnCreateViewHolder() is called automatically by the adapter each time it needs to display a data item.
When you inflate the xml layout file, you get a view, and then you wrap that in an instance of your ViewHolder class and return that object.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
    //create an instance of LayoutInflater
    val layoutInflater = LayoutInflater.from(parent.context)
    //inflate the view
    val itemViewHolder = layoutInflater.inflate(R.layout.list_item, parent,
false)
    return ViewHolder(itemViewHolder)
}
```

OnBindViewHolder is called each time the adapter needs to display a new data item. It passes in the reference to the ViewHolder and the position of the data item in the collection. The job of OnBindViewHolder is to take that data object and display its values.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    //get data at the position
    val item = myItemList?.get(position)
    //set the text of the textview to the name
    holder.itemTextView.text = item?.name ?: ""
}
```

getItemCount() returns the number of items in the collection.

```
override fun getItemCount(): Int {
    if (myItemList != null) {
        return myItemList!!.size
    } else return 0
}
```

Go into MainActivity.kt.
We need to define an instance of the view model so we can access a single source of truth for our data and we'll do that at the class level.

```
private val viewModel: ItemViewModel by viewModels()
```

In onCreate() we need to

1. set up our recyclerView
2. instantiate an adapter with our view model
3. set the adapter to the RecyclerView
4. set a Layout Manager for our RecyclerView instance
    a. A layout manager positions item views inside a RecyclerView and determines when to reuse item views that are no longer visible to the user.
    b. We're using a LinearLayoutManager which shows the data in a simple list — vertically or horizontally (by default vertically).
5. create the LiveData observer so that whenever the onChanged event is triggered we notify the recycler view that the data has changed.
6. assign the observer to our view model's list of items. So whenever this list is changed the onChanged event will fire and the observer will be called.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    setSupportActionBar(findViewById(R.id.toolbar))

    //get the recycler view
    val recyclerView: RecyclerView = findViewById(R.id.recyclerView)

    //divider line between rows
    recyclerView.addItemDecoration(DividerItemDecoration(this,
LinearLayoutManager.VERTICAL))
```

```
    //set a layout manager on the recycler view
    recyclerView.layoutManager = LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false)

    //define an adapter
    val adapter = MyListAdapter(viewModel)

    //assign the adapter to the recycle view
    recyclerView.adapter = adapter

    findViewById<FloatingActionButton>(R.id.fab).setOnClickListener {view ->
        ...
    }

     //create the observer
     viewModel.itemList.observe(this, Observer {
         // TODO update adapter
     })
}
```

You should be able to run the app but the recycler view will be empty.

RecyclerView Adapter Updates

When the data set that an adapter binds to a recycler view uses changes, the adapter needs to be notified so it can update the recycler view. The RecyclerView Adapter has methods to notify the adapter of changes to the data.

- notifyItemInserted(int pos)
  - Notify that a single item was inserted at a given position
- notifyItemRangeInserted(insertIndex, items.size())
  - Notify that multiple items were inserted starting at a given position
- notifyItemRemoved(int pos)
  - Notify that a single item was removed at a given position
- notifyItemRangeRemoved(startIndex, count);
  - Notify that multiple items were removed starting at a given position
- notifyItemChanged(int pos)
  - Notify that item at position has changed.
- notifyItemMoved(fromPosition, toPosition)
  - Notify that an item has moved to a new position
- notifyDataSetChanged()
  - Notify that the dataset has changed
  - Use only as last resort, especially with long lists

In MyListAdapter.kt we need to create a method that will update the array list using the view model and then notify the adapter that the data has changed.

```
fun update(){
    myItemList = itemViewModel.itemList.value
    notifyDataSetChanged()
}
```

In MainActivity.kt we will call this in our observer.

```
viewModel.itemList.observe(this, Observer {
    adapter.update()
})
```

Add Items

Let's use the FAB in the MainActivity class to add items to the list.

First in activity_main.xml change the fab to +

```
app:srcCompat="@android:drawable/ic_input_add"
```

Add these strings to your string resource file since we'll be using them in our alert dialog.

```
<string name="addItem">Add Item</string>
<string name="add">Add</string>
<string name="itemAdded">Item Added</string>
<string name="action">Action</string>
<string name="cancel">Cancel</string>
```

In MainActivity.kt update the onClickListener handler for the floating action button.
We'll use an alert dialog with an edittext for the user to enter an item.
We'll add the item to our list and then call the delete method in our view model class to update our view model. This will also trigger our observer so the onChanged event will fire and our itemObserver will be called.

```
findViewById<FloatingActionButton>(R.id.fab).setOnClickListener {view ->
    //create alert dialog
    val dialog = AlertDialog.Builder(this)
    //create edit text
    val editText = EditText(applicationContext)
    //add edit text to dialog
    dialog.setView(editText)
    //set dialog title
    dialog.setTitle(R.string.addItem)
    //set OK action

    dialog.setPositiveButton(R.string.add) {dialog, which ->
        val newItem = editText.text.toString()
        if (!newItem.isEmpty()){
            //add item
            viewModel.add(Item(newItem))
            Snackbar.make(view, R.string.itemAdded, Snackbar.LENGTH_LONG)
                .setAction(R.string.action, null).show()
        }
    }
    //sets cancel action
    dialog.setNegativeButton(R.string.cancel) { dialog, which ->
        //cancel
    }
```

```
    dialog.show()
}
```

The FAB should now let you add items in an alert dialog and then when the view model is updated the onChanged event is fired and the adapter updates the list in the recycler view.
The SnackBar is not really needed since you can see the item added to the view, I just decided to leave it in there.

Delete Items

We want an item to be deleted when the user does a long-press on an item. We'll do this though a context menu so they can choose to delete the item or not.
Add these strings to your string resource file since we'll be using them in our alert dialog.

```
<string name="delete">Delete?</string>
<string name="deleteItem">Item Deleted</string>
<string name="cancel">Cancel</string>
<string name="yes">Yes</string>
<string name="no">No</string>
```

In MyListAdapter.kt we'll update onBindViewHolder() to set up the long click listener and create the context menu.
When the user clicks on the "yes" menu we'll call the delete method in our viewmodel. This is why it was better to pass in the view model than the array of items from MainActivity.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    ...

    //context menu
    holder.itemView.setOnCreateContextMenuListener(){menu, view, menuInfo ->
        //set the menu title
        menu.setHeaderTitle(R.string.delete)

        //add the choices to the menu
        menu.add(R.string.yes).setOnMenuItemClickListener {
            //remove item from view model
            itemViewModel.delete(item!!)
            Snackbar.make(view, R.string.deleteItem, Snackbar.LENGTH_LONG)
                .setAction(R.string.action, null).show()
            true
        }
        menu.add(R.string.no)
    }
}
```

In testing the app make sure you test rotation as well. You'll notice that our list remains the same, and adding and deleting items continues to work, all without specifically implementing any methods to save state. This is part of the benefit of ViewModel, it is persistent across launches of our activity.