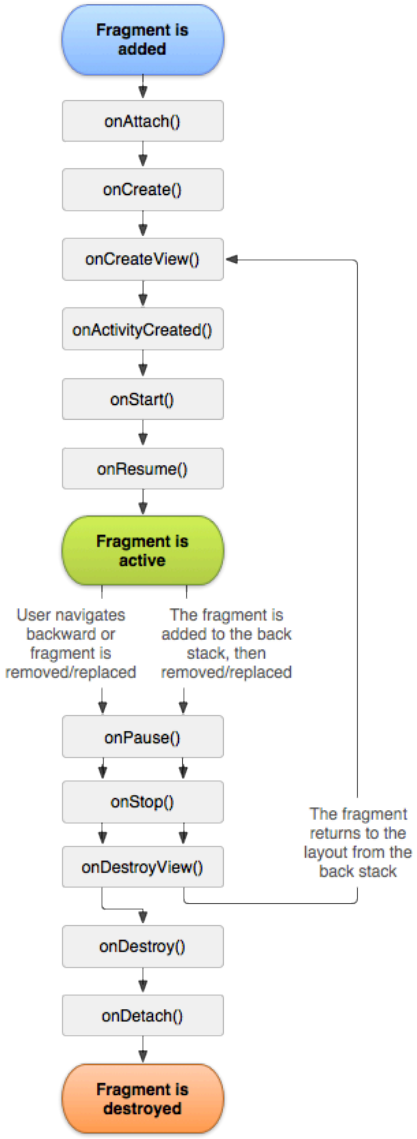
University of Stirling
Computing Science
Mobile App Development

Android Project: Fragments & ViewModel

In this session we will start the project off and focus on Android Fragments. The aim of this lab is to create an Android application which has two tabs, and two fragments. These two fragments will exchange some basic data via a ViewModel. Fragment 1 will have a textbox and a button, which then clicked will transfer the contents of the textbox to Fragment 2 for display.

Fragments

Fragments (https://developer.android.com/guide/fragments) allow for a more flexibile UI. By dividing the layout of an activity into fragments, you can modify the activity's appearance at runtime. For example one fragment will show input controls for users to select units, whereas the other will show a input panel for numeric user input. Users will be able to toggle between the two fragments by using Tabs in the application.



Each fragment has its own set of lifecycle callback methods and handle their own user input events. You can see them a bit like a subactivity. Fragments define their own layout and have their own lifecycle (albeit linked to the one of the activity). Fragments can belong to different activities. For instance on a tablet, two fragments belonging to an activity can be shown by the same activity, whereas on a phone, one fragment is shown by one activity and the second fragment by a second activity.

To create a fragment, you create a subclass of Fragment. The Fragment class has code that looks a lot like an Activity. You should implement the following methods for a fragment:
onCreate() The system calls this when creating the fragment.
onCreateView() The system calls this when it's time for the fragment to draw its user interface for the first time. To draw a UI for your fragment, you must return a View from this method that is the root of your fragment's layout. We will use this function in the application.
onPause() The system calls this method as the first indication that the user is leaving the fragment

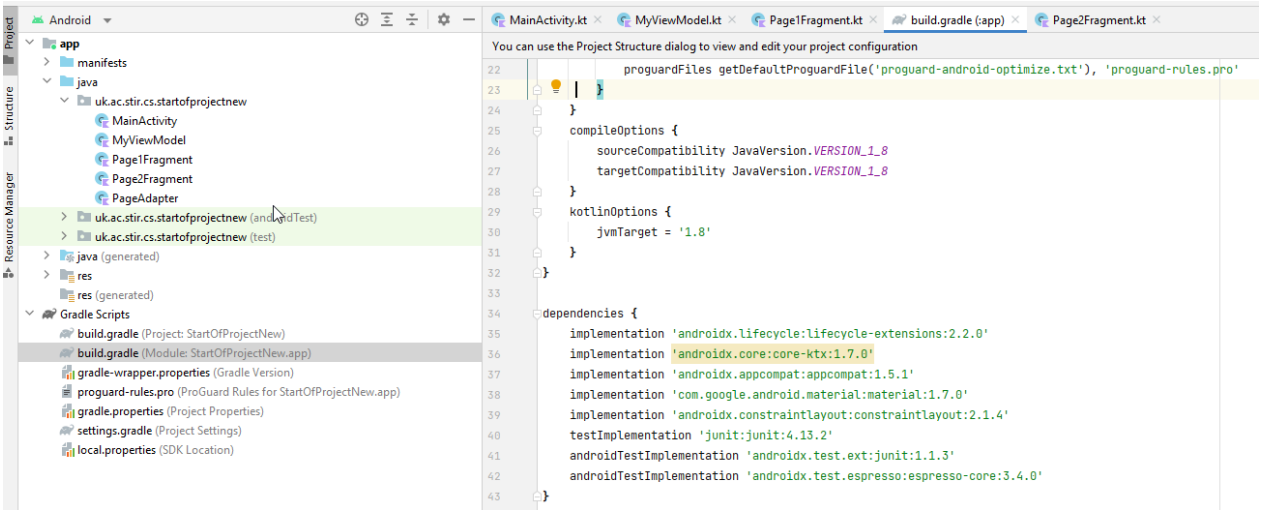
ViewModel

Fragments are meant to be self-contained and define their own layout and behavior. However, many applications require communication between an activity and its fragments or between two or more fragments. To keep fragments self-contained, fragments should **not** communicate directly with other fragments or with their host activity. To share (persistent) data using a ViewModel is recommended (https://developer.android.com/guide/fragments/communicate). ViewModels (https://developer.android.com/topic/libraries/architecture/viewmodel) are often used with LiveData (https://developer.android.com/topic/libraries/architecture/livedata) or rather MutableLiveData which defines additional public methods (https://developer.android.com/reference/android/arch/lifecycle/MutableLiveData).

Implementing the app

Let's start the project. Create a new Android Studio Project. We will start with the layout. As we will use some additional components, you will need to add a couple of dependencies to the Gradle script for your module:

```
implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
implementation 'com.google.android.material:material:1.7.0'
```



Then in the main layout we will add three components. You can chose to base your layout on the LinearLayout (more straightforward) or ConstraintLayout (default, many more options). We will need to add a Toolbar, TabLayout, and a ViewPager2. The toolbar will eventually host a menu.

Add a Toolbar, TabLayout (don't confuse with TableLayout!) and ViewPager2 element (all in Containers) to your layout (If you have a TextView by default this is best removed). As before, this step is quite fiddly, but considerably easier with a LinearLayout. We do not need the 3 TabItems which may be automatically generated with the TabLayout. You can delete them. Try and arrange these components starting at the top of the screen (using the grid). Now we can edit the look and feel of these components as follows:

Toolbar:

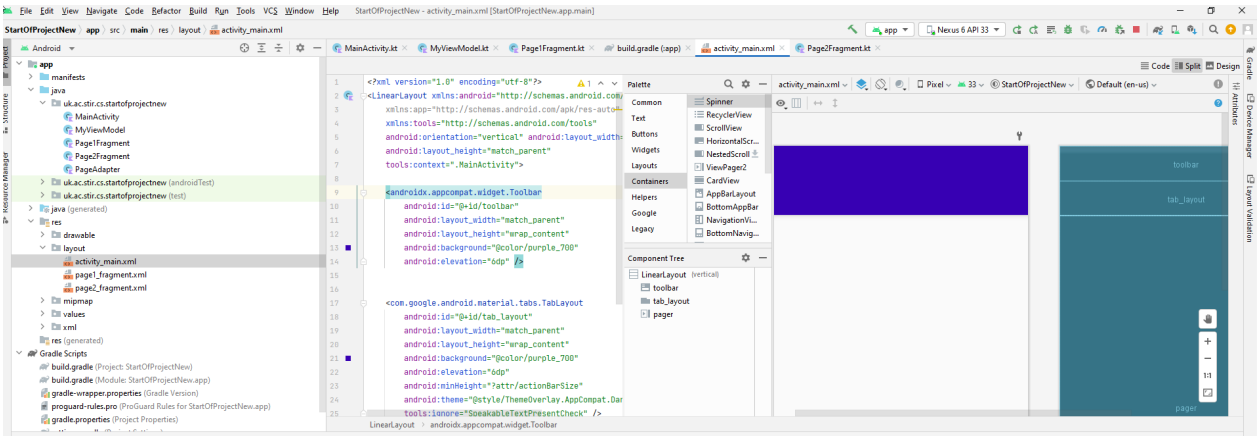
```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/purple_700"
    android:elevation="6dp" />
```

TabLayout:

```
<com.google.android.material.tabs.TabLayout
    android:id="@+id/tab_layout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/purple_700"
    android:elevation="6dp"
    android:minHeight="?attr/actionBarSize"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"/>
```

ViewPager2:

```
<androidx.viewpager2.widget.ViewPager2
    android:layout_width="match_parent"
    android:layout_height="fill_parent"
    android:id="@+id/pager" />
```



If you are using ConstraintLayout as the base layout you will need to add constraints for these three components as follows:

For the Toolbar (paste at the end of the Toolbar tag definition, before the closing tag):

```
app:layout_constraintBottom_toTopOf="@+id/tab_layout"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
```

For the TabLayout (paste at the end of the TabLayout tag definition, before the closing tag):

```
app:layout_constraintBottom_toTopOf="@+id/pager"
app:layout_constraintEnd_toEndOf="parent"
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toBottomOf="@+id/toolbar"
```

For the ViewPager (paste at the end of the ViewPager tag definition, before the closing tag):

```
app:layout_constraintBottom_toBottomOf="parent"
app:layout_constraintStart_toStartOf="@+id/tab_layout"
app:layout_constraintTop_toBottomOf="@+id/tab_layout"
```

Next we need a layout for each of the two fragments. Create two new layouts (under resources/layout, named page1_fragment and page2_fragment (or similar). Besides the default layout format (it might suggest ConstraintLayout, but LinearLayout is sufficient), for the page1 layout, add an EditText (with android:id="@+id/editValue") and a Button (say with android:id="@+id/vButton"). For the page 2 layout, we just need a TextView (with android:id="@+id/textView").

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <EditText
        android:id="@+id/editValue"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="textPersonName"
        android:text="Message" />

    <Button
        android:id="@+id/vbutton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Update" />

</LinearLayout>
```

```
umPad - version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Fragment 2" />

</LinearLayout>
```

Finally, as we have defined our own Toolbar, in the themes.xml file (under values/themes) change the style of the app to (keep the name as suggested, just change the parent):

```
<style name="Theme.StartOfProjectNew" parent="Theme.AppCompat.Light.NoActionBar">
```

This defines the required layout for the basic app skeleton. Next we move to the code.

Firstly, lets create a class for the ViewModel which will act as the medium to exchange information between the two Fragments. The ViewModel code is fairly basic: it's a class which extends ViewModel and a single internal variable which is MutableLiveData. There is then just a function for access and initialisation.

```
class MyViewModel : ViewModel() {
    private val _value = MutableLiveData<String>()

    val value: MutableLiveData<String>
        get() = _value

    init {
        _value.value = "default"
    }

}
```

Next let's move to the two fragments. The class for each Fragement will extend the base class Fragement. Within these classes, you will need to overload the standard method onCreateView(). This gets executed when the Fragement comes into view. For the first Fragment, the method should link to the ViewModel defining its scope to be the Activity of the app (so that both Fragments can "see" it), load the layout for the fragement, link the contents of the EditText to the ViewModel and return the View.

Below is sample code for Fragement 1:

```
class Page1Fragment : Fragment() {
    lateinit var viewModel: MyViewModel

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
savedInstanceState: Bundle?): View? {
        viewModel = activity?.run {
            ViewModelProvider(this)[MyViewModel::class.java]
        } ?: throw Exception("Invalid Activity")

        val view = inflater.inflate(R.layout.page1_fragment, container, false)
        val but = view.findViewById<Button>(R.id.vbutton)
        val value = view.findViewById<EditText>(R.id.editValue)
        but.setOnClickListener {
            viewModel.value.setValue(value.getText().toString())
        }
        return view
    }
}
```

Fragement 2 will also need to link to the ViewModel, load its layout, "observe" the value in the ViewModel and output any changes and finally return the View.

```
class Page2Fragment : Fragment() {
    lateinit var viewModel: MyViewModel

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
savedInstanceState: Bundle?): View? {
        viewModel = activity?.run { ViewModelProvider(this)[MyViewModel::class.java]
        } ?: throw Exception("Invalid Activity")

        val v = inflater.inflate(R.layout.page2_fragment, container, false)
        val valueView = v.findViewById<TextView>(R.id.textView)

        val valueObserver = Observer<String> { newValue ->
            valueView.text = newValue.toString()
        }
        viewModel.value.observe(viewLifecycleOwner, valueObserver)

        return v;
    }
}
```

Android Studio will suggest a number of imports for both Fragments.

Next, you will need to create a class for the FragmentStateAdapter https://intensecoder.com/android-swipe-fragments-with-viewpager2-in-kotlin/. This class will manage the two fragments and their status. Below is some sample code for the adaptor. Besides the constructor, there is a method getItem() which returns an instance of the correct fragement

based on an integer input indicating the tab position. Similarly, there is a method which returns the number of fragements managed by this class. This class also needs a few imports.

```
class PageAdapter(fa: FragmentActivity, private val mNumOfTabs: Int) :
    FragmentStateAdapter(fa) {

    override fun getItemCount(): Int {
        return mNumOfTabs
    }
    override fun createFragment(position: Int): Fragment {
        return when (position) {
            0 -> Page1Fragment()
            1 -> Page2Fragment()
            else -> Page1Fragment()
        }
    }
 }
```

Finally, we will need to put it all together in the main app Activity class. Here we will need to overwrite the onCreate() method (as you have done in previous practicals. We will need to set the View (main activity layout with the toolbar and tabs etc). Then we need to initialise the toolbar by loading its resource and passing it into a Toolbar object. This toolbar object then needs to be linked with the main activity.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_layout_linear)
        val toolbar = findViewById<Toolbar>(R.id.toolbar)
        setSupportActionBar(toolbar)

        ... //include code below
}
```

Next we will need setup the two tabs we need for the fragments. For this we need to load the TabLayout resource and define the two tabs. The code below uses two String resources which contain strings to identify the tabs. You may want to create these two String resources also, or alternatively, simply use a string.

```
val tabLayout = findViewById<TabLayout>(R.id.tab_layout)
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_page1))
tabLayout.addTab(tabLayout.newTab().setText(R.string.tab_page2))
tabLayout.tabGravity = TabLayout.GRAVITY_FILL
```

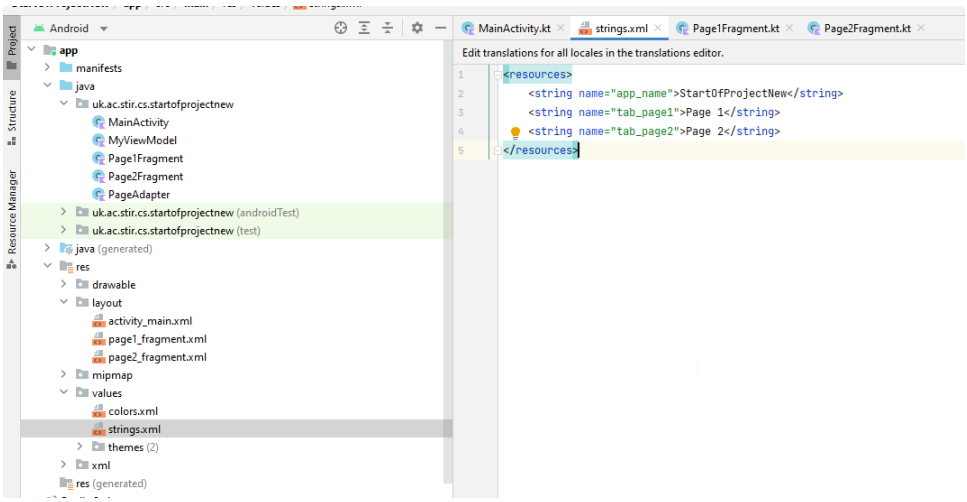
Then link the ViewPager2 area in the layout with an instance of the adaptor class defined above.

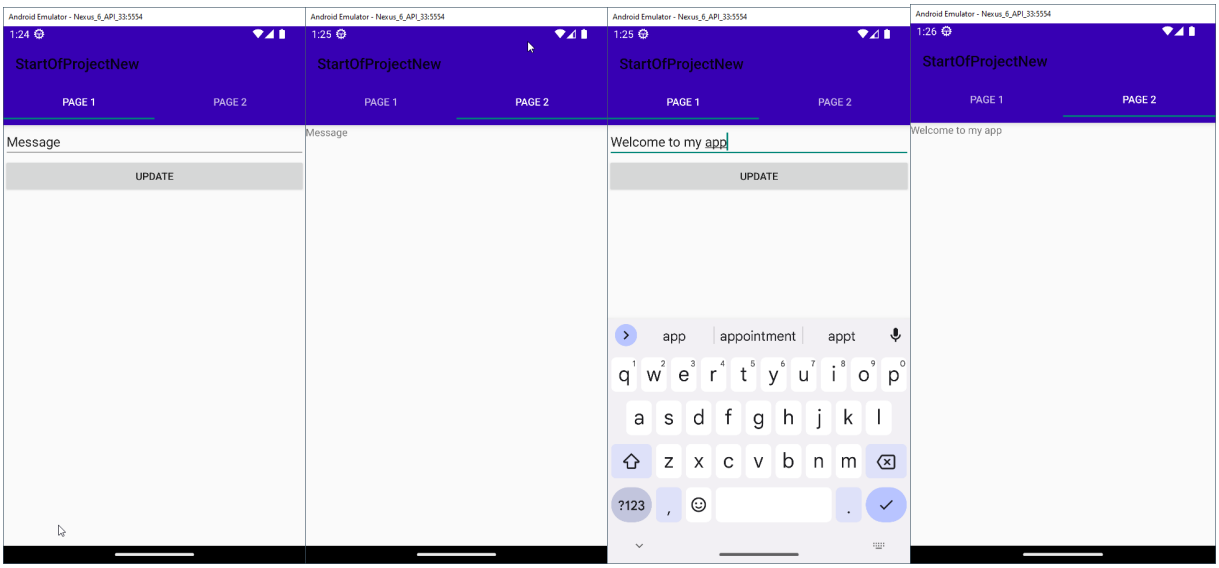
```
        val viewPager = findViewById<ViewPager2>(R.id.pager)
        val adapter = PageAdapter(this,2)
        viewPager.setAdapter(adapter)

        TabLayoutMediator(tabLayout, viewPager) {
                tab, position -> tab.text = "Page " + (position + 1) }.attach()
    }
}
```

What's left to do is to include the required imports.

Then define the two Strings tab_page1 and tab_page2 in the strings.xml file resources/values. They contain the strings for the Tabs. Then try your app.





You are now ready to start your Assignment. Before that, you may want to try and store other data types in the ViewModel.