Lab 3 - Fragments, Advanced UI Elements

In this assignment, you will create an app that combines Activities and Fragments, and organize the Fragments in tabs. In addition, we are going to see advanced UI elements: FloatingActionButton & Snackbar, and CardView & RecyclerView.

A fragment is a self-contained component with its own UI and lifecycle; it can be reused in different parts of an application's user interface depending on the desired UI flow for a particular device or screen. According to the Android developer's website: "A Fragment represents a behavior or a portion of user interface in an Activity. You can combine multiple fragments in a single activity to build a multi-pane UI and reuse a fragment in multiple activities. You can think of a fragment as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a "sub activity" that you can reuse in different activities)."

The easiest way to start understanding this is to see Fragments in action. In this lab, we will build a rough sketch of an app for astronomer citizen scientists, consisting of one activity and three fragments. "Citizen science" is scientific research conducted, in whole or in part, by amateur (or nonprofessional) scientists: members of the general public often contribute to science with their observations, data processing, etc.

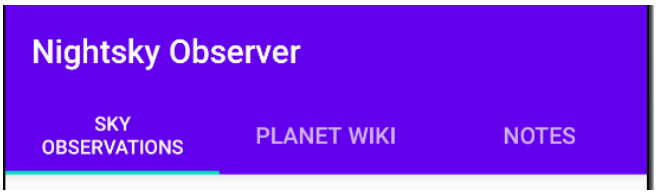
The app will have three sections:

- *Sky observations* - where timestamped observations are collected
- *Planet Wiki* - that provides information about planets
- *Notes* - where a citizen scientist can take notes

We are going to implement a rather advanced application in a short period of time, thus, we won't start from zero. Open a new project from GitHub https://bitbucket.org/pbdfrita/pbd2023-lab-3

Fragments in Tabs

Examine the code. You should see the **MainActivity**, three Fragment classes (**Tab1Fragment**, **Tab2Fragment**, **Tab3Fragment**), and a **RecyclerAdapter**. Our **MainActivity** should host a placeholder for the Fragments and a bar that will let a user navigate among tabs. Each tab should be a Fragment, something like this:



What you see in this image is a larger **AppBarLayout** that contains a **Toolbar** (the part with the title) and a **TabLayout** (with tabs). Open `activity_main.xml` and add a `com.google.android.material.tabs.TabLayout` element just below the **Toolbar**. Set its ID to `tab_layout`, `app:tabMode` to `"fixed"`, and `app:tabGravity` to `"fill"`.

Note that this **TabLayout** will only render the bar, but will not actually cycle through the content. We need another element for that. The element is **ViewPager2** - go ahead and add it in `activity_main.xml` just under the **AppBarLayout**. Your **ViewPager2** should stretch as much as

possible in width and height (use `match_parent`) and should have the following property `app:layout_behavior="@string/appbar_scrolling_view_behavior"`

Stop for a moment to analyse your work. You have a **TabLayout** where the tab title should be shown and where a user should make a selection. You also have a **ViewPager2**, think of it as a frame where your Fragments can go. Now you need to connect the **TabLayout** with the **ViewPager2** and the Fragments.

To connect the fragments with the **ViewPager2** we need to use an adapter, in particular **FragmentStateAdapter** (from `androidx.viewpager2`). Create a new class called **TabPagerAdapter** and make it extend **FragmentStateAdapter**. We will need to make use of the class constructor. In Kotlin, there are 2 types of constructors:

- Primary constructor - concise way to initialize a class
- Secondary constructor - allows you to put additional initialization logic

Here we will be using the primary constructor - which in Kotlin is part of the class header:

```
class TabPagerAdapter(fa: FragmentActivity?, private val tabCounter: Int) : FragmentStateAdapter(fa!!)
```

This notation is equivalent to calling `super(fa)` and setting a private class member `tabCounter`.

This class needs to implement `createFragment` and `getItemCount` functions, so go ahead and override them. The `createFragment` function takes an integer that indicates a position that a **ViewPager2** is "looking at" and returns the appropriate Fragment. Create a *when* statement (Kotlin enhanced version of "switch") that will, depending on the position (0, 1, 2), return either `Tab1Fragment()`, `Tab2Fragment()`, or `Tab3Fragment()`.

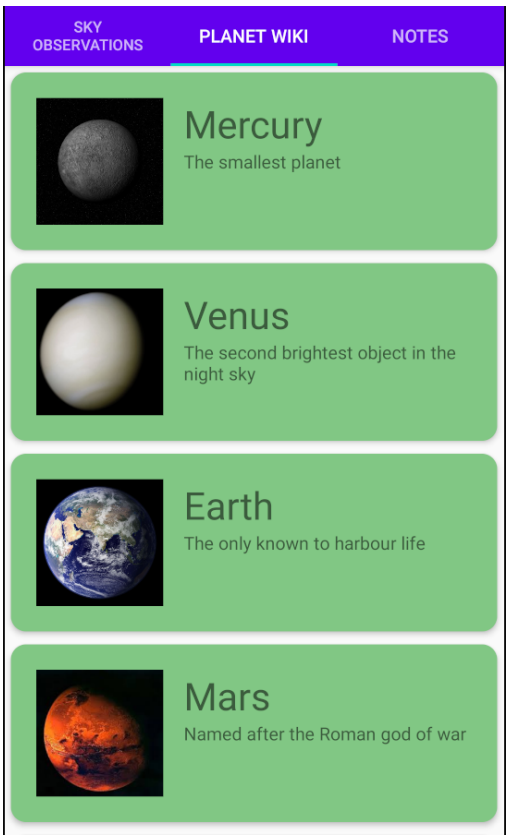
The `getItemCount` function should simply return the number of items (tabs), so go ahead and return `tabCounter`.

We have our fragments connected with the **ViewPager2**, so now we need to connect the **ViewPager2** and the **TabLayout**. Go to **MainActivity** and implement the `configureTabLayout` function. First, create references to the **TabLayout** and the **ViewPager2** (use `findViewById()`, or better yet, this is a good opportunity to try Android's view binding). Then, create a new **TabPagerAdapter** that takes "`this`" and `NUM_OF_TABS` as arguments. Set the **ViewPager2's** adapter to the newly created adapter via ViewPager2's `.setAdapter` method. To finally connect the **TabLayout** and **ViewPager2,** create a new **TabLayoutMediator**. It should look like this:

```
TabLayoutMediator(tabLayout, viewPager

) { tab, position ->

// You should set tab titles here

}.attach()
```

The **TabLayoutMediator** will connect each **Fragment** in the **ViewPager2** and will call `onConfigureTab`. Here you should set the titles for the tabs. Use a when statement operating on "`position`" and `tab.setText()` to set the title String (e.g. "Sky Observations", etc.)

CardView & RecyclerView



Remember how Android destroys your views when an Activity goes in the background? Of course, it does this to preserve memory. However, you could still use a lot of memory if you have a lot of items in your ListView. Not all of these items can be shown on the screen at the same time, so perhaps it makes more sense to dynamically create the items as a user is scrolling along the list. This is exactly why we have the **RecyclerVIew**!

We will start coding the second Fragment first; here we are going to use a **RecyclerView** to list cards with info about different planets. The cards will be implemented using the **CardView**.

We have already added **RecyclerView** for you - open `fragment_tab2.xml` to check it and get familiar with it. A **RecyclerView** uses an Adapter to get connected with the data. Open **RecyclerAdapter** class. This is a partly implemented class that connects planet information with the **RecyclerView**.

As a user scrolls through the information in a **RecyclerView**, the adapter creates **ViewHolders** and populates them with information when `onBindViewHolder` is called. Here we are extending the default **RecyclerView.ViewHolder** and implementing the one that shows cards that can be clicked on. Notice an inner class **CardViewHolder** that extends **RecyclerView.ViewHolder**.

But how does the **CardViewHolder** know *how* to render the information? In `card_layout.xml` we have defined the layout of a single card - please check it. Then, in `onCreateViewHolder` use the following to inflate the card view:

```
LayoutInflater.from(viewGroup.context).inflate(R.layout.card_layout,
        viewGroup, false)
```

The **View** returned by the above call should be used to instantiate a new **CardViewHolder**, that should then be returned from the function.

This makes sure we have a **CardViewHolder** created when needed, but to connect it with the data in each new **CardViewHolder** we need a reference to an image, title, and detail text. In the **CardViewHolder** constructor set the `itemImage`, `itemTitle`, and `itemDetail` to refer to the corresponding views from `card_layout.xml`. Finally, to show a **Snackbar** every time a user clicks on an item, call `setOnClickListener` method of `itemView` and show a **Snackbar**, just like in the first fragment. In the shown text state the position of the card by using `getAdapterPosition()`.

The only thing we are left with is setting the actual data. This happens in `onBindViewHolder`. Use the above defined references `itemTitle`, `itemDetail`, and `ItemImage` of `viewHolder`, and their `setText` or `setImageResource` methods to set the values to `titles[i]`, `details[i]`, and `images[i]`.

This is a good time to test your app and debug any issues.

FloatingActionButton and Snackbar

In the first Fragment we have a **ListView** (confirm that in `fragment_tab1.xml`). This list is going to be populated with timestamps of planet observations when a user clicks on a floating button. In addition, the user will be shown info that an item has been added to the list and will have an option to undo the action. The picture below should give you an idea:

The floating button should be implemented via **FloatingActionButton** from `com.google.android.material`. In `fragment_tab1.xml` add the button right after the **ListView**:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
      android:id="@+id/fab"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="bottom|end"
      android:layout_margin="@dimen/fab_margin"
      app:layout_constraintBottom_toBottomOf="parent"
      app:layout_constraintEnd_toEndOf="parent"
      app:srcCompat="@drawable/ic_add_entry" />
```

Go to **Tab1Fragment**. Here you already have your **ListView** connected with an **ArrayAdapter**. This adapter defines that an **ArrayList** of **Strings** called `listItems` is shown in `myListView` and that `android.R.layout.simple_list_item_1` is used to define the rendering of each item. Further, we have already implemented a function `addListItem` that adds an observation to the list.

To define what happens when a user clicks on the floating button, in `onCreateView` find your **FloatingActionButton** (use `findViewById` on "`view`", or use view binding) and set an **OnClickListener** (just like you did with any other button) and there call `addListItem()`. Furthermore, add the following line to show a snackbar to the user:

```
Snackbar.make(v, "Item added to list", Snackbar.LENGTH_LONG)

        .setAction("Undo", undoOnClickListener).show()
```

Notice that the snackbar also sets an action, called Undo, on which an `undoOnClickListener` is called. This we have already implemented for you, but please check it and make sure you understand it - it removes the newly added item.

Go ahead and test your app now. You should be able to add observations to the list and to remove them if you click on the Undo action in the Snackbar.

If you were not present in the lab for solving this assignment you should commit your solution to a private repository named **PBD2023-LAB-3** in your Bitbucket account. User **pbdfrita (pbdfrita@gmail.com)** must be added as a read-only member of this repository. The code must be committed by Sunday (March 12th) at 23:59.

Happy coding!