IMD 4008 / Mobile User Interfaces

Tutorial 4 – Navigation via Fragments

Objectives

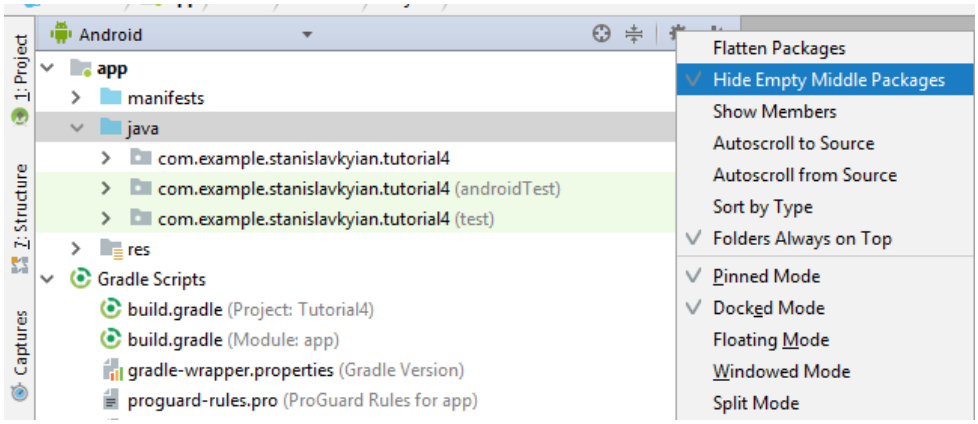
1. Develop an app that allows changing between screens
2. Use the Bottom Navigation View bar to support multiple different views
3. Add fragments for each Navigation Bar option, and use the ViewModel to pass data between them (e.g., for saving settings in one view that affect another view).

Introduction

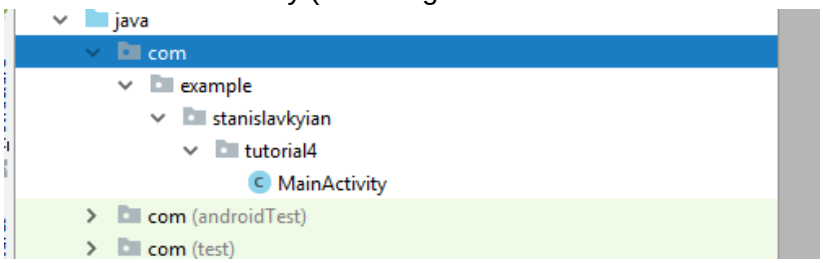
1. Creating the Bottom Navigation Activity

Start by downloading tutorial4_base.zip from CULearn, and storing it where you've been saving your Android Studio projects. Unzip it, and open Android Studio. Open the project by selecting "Open Existing Project" and opening the project directory, which by default will be Tutorial4_base\Tutorial4. (Note: D**o not create the project from scratch, as we have done in previous tutorials**, to avoid some version problems later, use the version we have provided).

You will likely be prompted to change the SDK location, click OK. The current package name is com.example.stanislavkyian.tutorial4. Let's first update this to use your name. From the project view, select your java folder, and click the gear icon. Un-check the "Hide Empty Middle Packages" option, seen below.



This should expand the folder hierarchy (reflecting the folder structure on the disk). See below:



Next, open the build.gradle (Module:app) file. Find the line “applicationId”. Modify this to replace “stanislavkyian” with “yourname” (putting your actual name instead of “yourname”). You will be prompted to resync your project: click the Sync Now button.

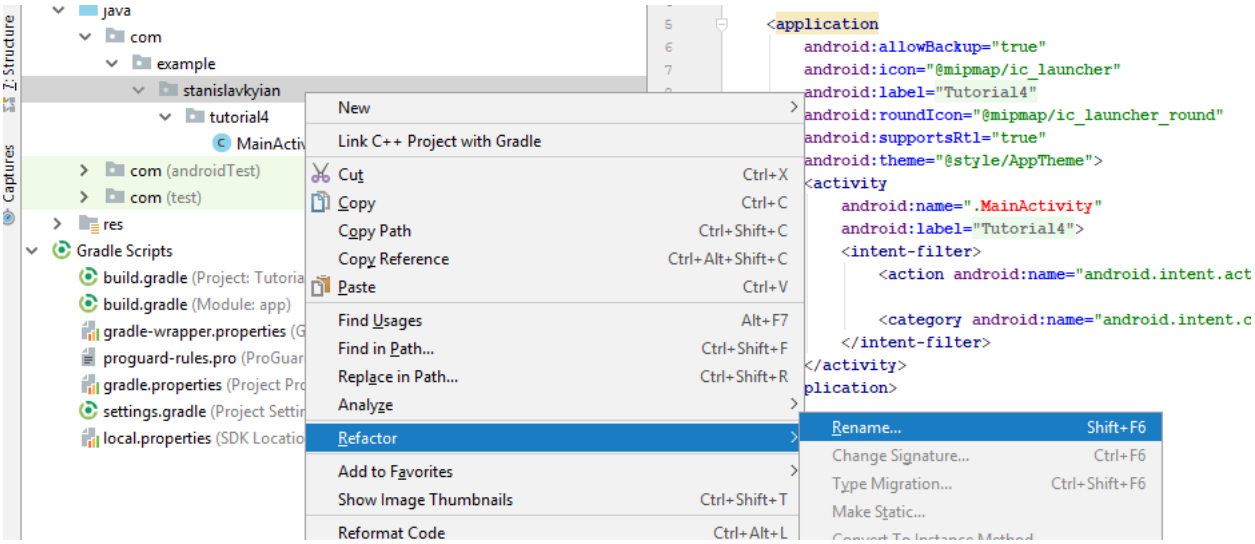
```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28
    defaultConfig {
        applicationId "com.example.stanislavkyian.tutorial4"
        minSdkVersion 16
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
```

Next, we modify the app manifest. This can be found under the project view under app > manifests > AndroidManifest.xml. Open this file and fine the line package=”com.example.stanislavykyian.tutorial4”. Once again, replace “stanislavkyian” with “yourname”.

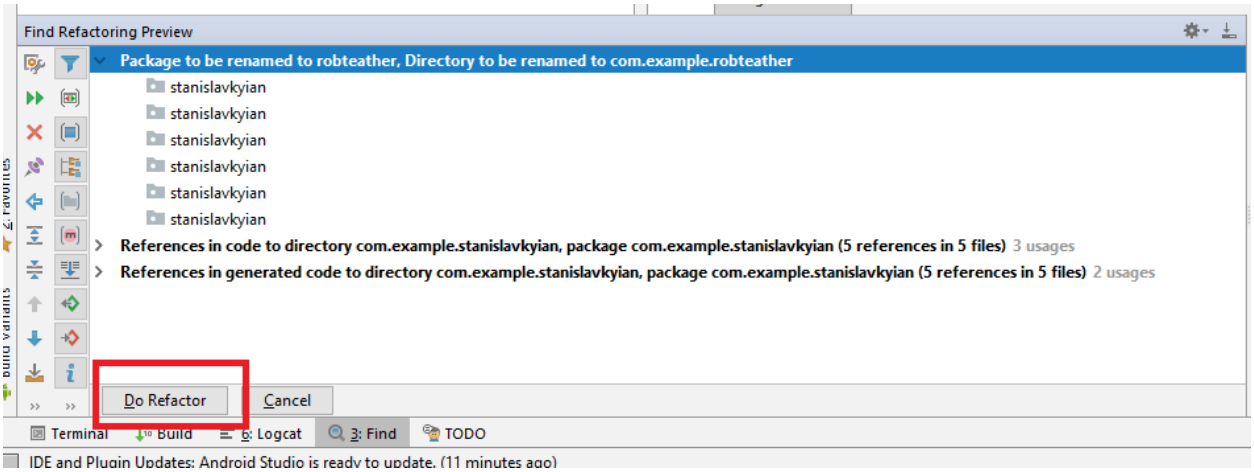
AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.stanislavkyian.tutorial4">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Tutorial4"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="Tutorial4">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Finally, back in the project view right click on the “middle” folder of your project hierarchy, the one that lists “stanislavkyian” as the name. From the menu, select Refactor > Rename, then “Rename package”. When prompted, update the name to “yourname” again. Hit the refactor button.

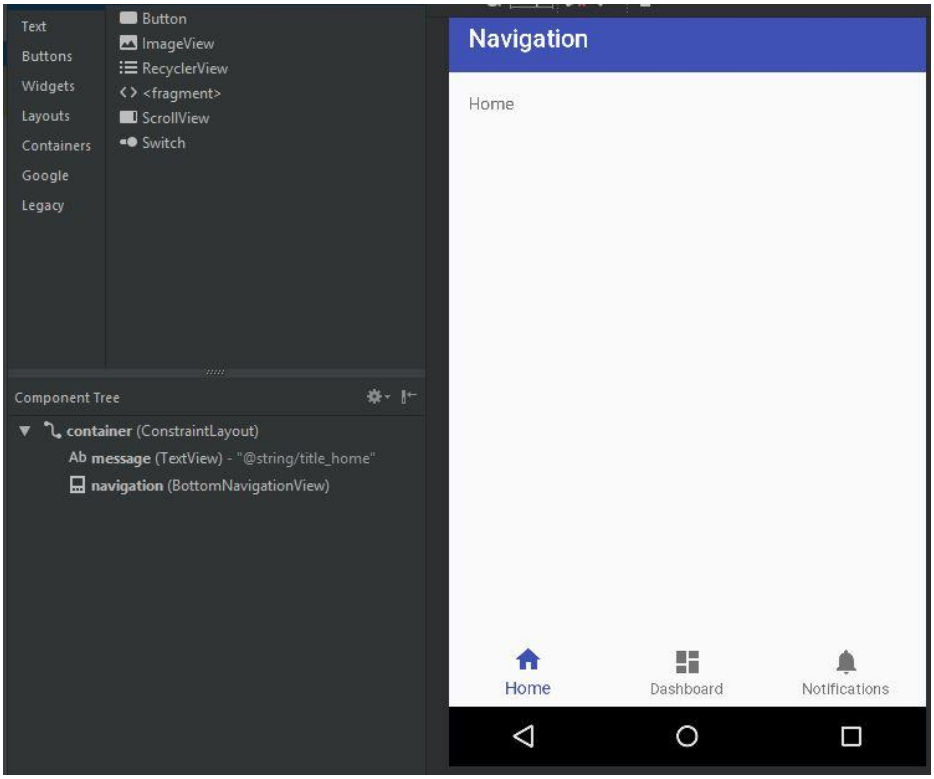


You will get a refactor “preview”, in the event log at the bottom of the screen. Click the “Do Refactor” button.



At this point, click the Build menu (top of the screen) and select “Clean Build”, then “Rebuild”. your project should now build (you can compact empty middle packages again in the project view). Your project should now build and be runnable in the emulator, as usual.

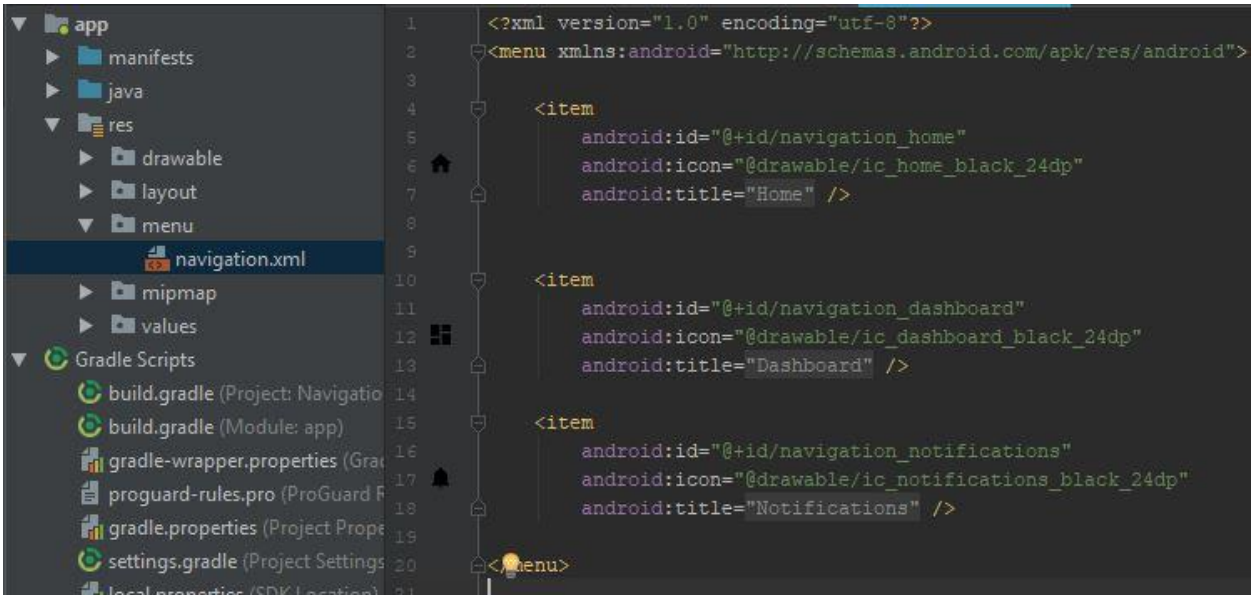
Once your project loads and builds, let’s first look at what has been generated for us. Head over to your activity_main.xml and Once look at the design panel, notice how a default BottomNavigationView (a navigation bar) was automatically created.



Now head over to the text panel of your xml and see the navigation component that was created.

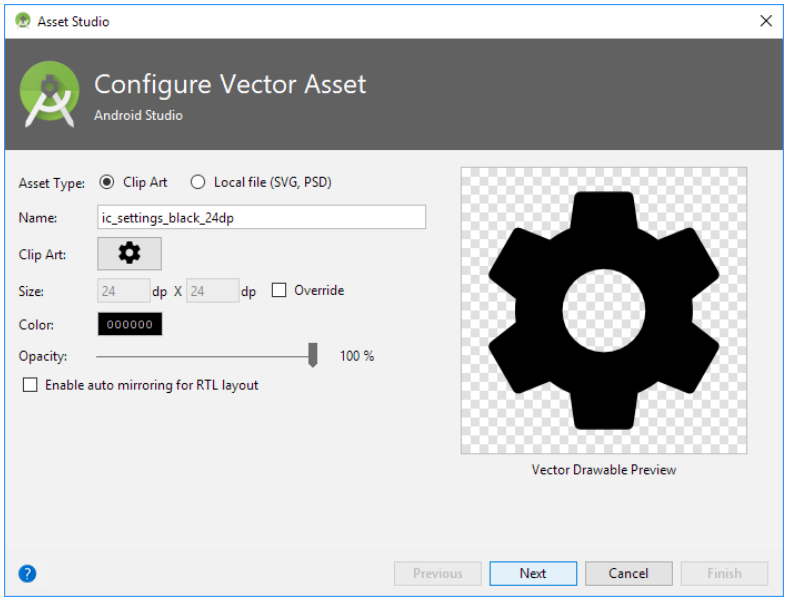
```
<android.support.design.widget.BottomNavigationView
    android:id="@+id/navigation"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="0dp"
    android:layout_marginStart="0dp"
    android:background="?android:attr/windowBackground"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:menu="@menu/navigation" />
```

Take note of the last parameter here “app:menu”, this is where we specify the options that our navigation bar should have, by default it is using “@menu/navigation”. Similar to strings and colours in previous tutorials, this is pointing it to another XML resource file found under “app/res/menu/navigation.xml”. Open it to see the structure of this file:

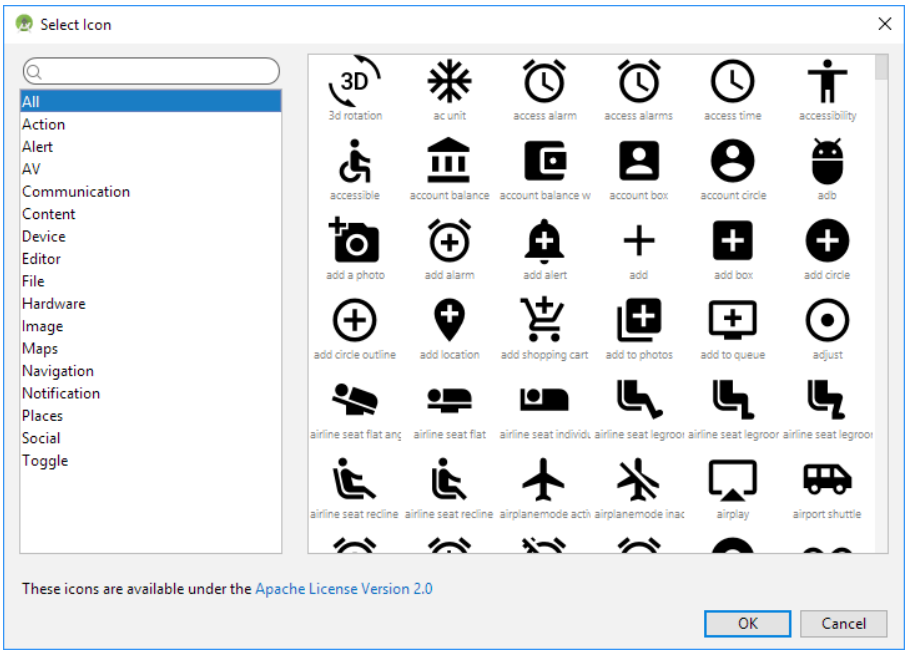


Each item in this file is a new option on the navigation bar, with an ID, icon, and title. As usual, the ID is used to reference each option, the title is the text that is shown in the UI. The icon is the image shown on the navigation bar for each option in the UI. Let's add a new option with a custom icon.

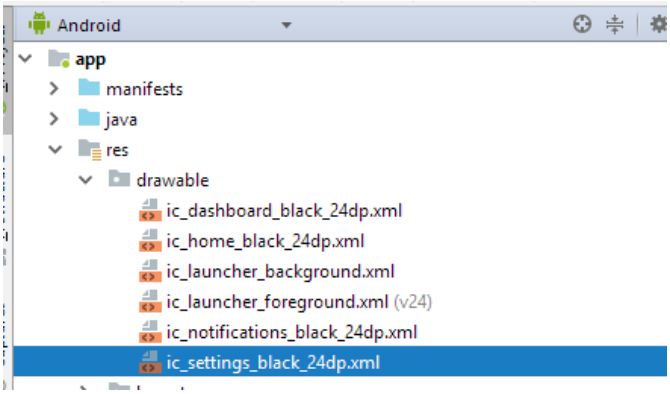
First let's add a new icon to our project. We can do this by adding one of the built-in icons. Right click your app > res > drawable folder. Select "New > Vector Asset". You should be presented with a dialog similar to the following:



Note that you can select local files (for custom icons) by selecting "Local File". For now, we will us a built-in icon. Click the "Clip Art" button. You will get a dialog that includes a lengthy list of built-in icons:



Using the search bar, find “Settings” and pick the black gear icon. Hit “OK”. You can leave its name as-is (ic_settings_black_24dp) or change it to something else (noting you will have to modify subsequent steps if you rename it). Finally, hit “Next” to finish adding the icon. It appear as a new resource under your drawables folder.



Now, go back to navigation.xml. Following the format seen with the other <item> tags, add a new item, with ID “navigation_settings” and title “Settings”. Note that you should (as usual) add a new string to your project in strings.xml for the Settings title, rather than directly entering it. Try to recall how to do this from previous tutorials, but ask for help (or just hardcode the string) if you get stuck. Finally, specify that the icon should be “@drawable/ic_settings_black_24dp” which will fetch our image from that resource folder. You should see the gear icon appear in the margin indicating success.

```
19      <item
20          android:id="@+id/navigation_settings"
21          android:icon="@drawable/ic_settings_black_24dp"
22          android:title="@string/title_settings" />
23
```

At this point, you can view your activity_main.xml Design view. You should see the gear icon added to the BottomNavigationView. Running the application will not (yet) produce the same behaviour for the new icon as the others, however.

To get the new navigation option working properly, open MainActivity.java file. Here you will see the event handler BottomNavigationView.OnNavigationItemSelectedListener. Inside the listener, there is pre-generated method called onNavigationItemSelected. This method is called any time the user selects one of the different BottomNavigationView options. Currently, this method just changes a TextView to show which option was clicked. You can see the pre-generated navigation options (Home, Dashboard, and Notifications). Following this model, add a new case for the Settings option:

```
@Override
public boolean onNavigationItemSelected(@NonNull MenuItem item) {
    switch (item.getItemId()) {
        case R.id.navigation_home:
            mTextMessage.setText("Home");
            return true;
        case R.id.navigation_dashboard:
            mTextMessage.setText("Dashboard");
            return true;
        case R.id.navigation_notifications:
            mTextMessage.setText("Notifications");
            return true;
        case R.id.navigation_settings:
            mTextMessage.setText("Settings");
            return true;
    }
    return false;
}
```

Try running your app and selecting each navigation option. At this point, each should respond in the same way, animating a transition to show which item is selected, and updating the text on the screen to indicate which option is active.

2. Adding Fragments

So far the navigation bar is working, but just changes the text on a TextView widget. For proper navigation, we want something more complex than this: we would like a new activity with its own Layout and Views to open up when clicking each navigation bar option. To do this, we will be using Fragments. In short, fragments are similar to activities, except they can be used as a portion of your UI in the same Activity, instead of having to change the activity overall.

Let's start by deleting the default TextView that was created with our app, followed by deleting the references to it in our MainActivity.java (which are called *mTextMessage*). We will replace the TextView with a FrameLayout component. The FrameLayout is where we will load the Fragments each time a new navigation option is selected. In your activity_main.xml file, inside the ConstraintLayout, but before the BottomNavigationView, add the following:

```
<FrameLayout
    android:id="@+id/fragment_container"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginBottom="56dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    />
```

We will set the ID to “fragment_container”. The layout_constraints will constrain the layout to the edges of our parent ConstraintLayout As a side note, this method of modifying constraints is often more convenient and more precise than dealing with the constraint handles in the Design view. The layout_marginBottom is set to 56dp since this is the height size of the navigation bar. This gives space to ensure the FrameLayout doesn’t overlap the BottomNavigationView bar.

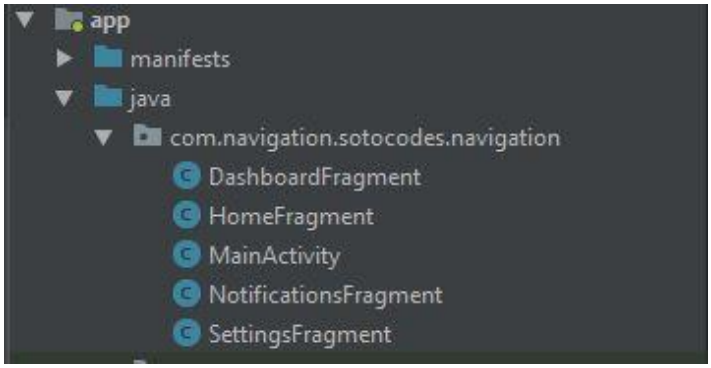
Since each Fragment will have its own layout, we need to add a new XML layout file for each of our navigation options. Head over to the layout folder under app/res/layout, right click the folder and select new->layout resource file. Name it “home_fragment”. Create three more layout files the same way, naming them “dashboard_fragment”, “notifications_fragment”, and “settings_fragment”. For these four layout files, we will create a very simple UI with just enough content to differentiate between each Fragment when they are running. Each layout should consist of two TextViews and one Button. In the end, each fragment should look like the following (note: the settings_fragment.xml file is shown here):


```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="100dp"
    android:text="Settings"
    android:textColor="#000000"
    android:textSize="24sp"
    android:textStyle="bold" />

<TextView
    android:id="@+id/settings_counter"
    android:layout_width="wrap_content"
    android:layout_gravity="center"
    android:layout_height="wrap_content"
    android:text="0"
    android:textColor="#000000"
    android:textSize="24sp"
    android:textStyle="bold"/>

<Button
    android:id="@+id/settings_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="20dp"
    android:text="Count" />
```

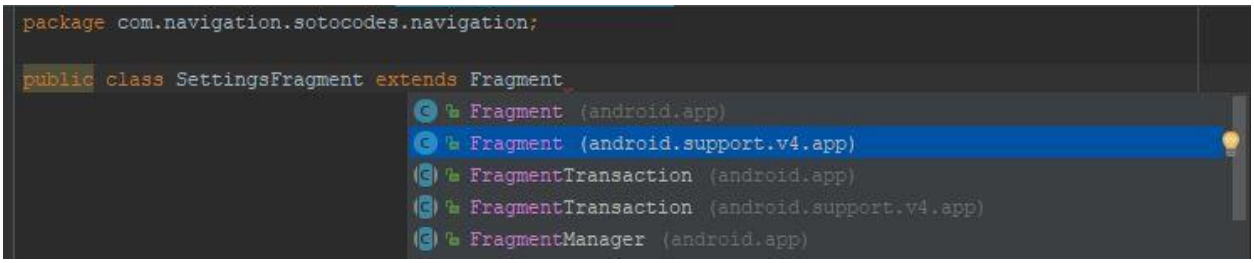
Notice the components with the arrow. These are the only components that should differ in each of the four Fragments’ layouts. The first TextView’s text should be the name of the navigation option (“Home”, “Dashboard”, “Notifications”, or “Settings”). The second TextView should be named after the navigation option + “_counter”. The Button should be named similarly: the navigation option + “_button”. The settings_fragment.xml code is shown above. Modify the rest

accordingly. Finally, you will likely have to specify constraints in each XML file to get the Views to proper positions (this is not shown in the above XML for clarity, but is seen in the design).

Just like any other layout file, we need an accompanying java class for each file. Head over to your Java folder, which currently contains only MainActivity.java. Right click on the folder and create a new Java class for each of the four components:



Head over to one of them, and add “extends Fragment” after the class name. This makes the class a subclass of Fragment, so it inherits the properties of Fragments we will use in the subsequent steps. When prompted to import the Fragment library, make sure you select the **support.v4.app** version which ensures backwards compatibility.



Now, we must implement the *onCreateView* method required by the Fragment class, add the following lines of code:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
             Bundle savedInstanceState) {
   return inflater.inflate(R.layout.dashboard_fragment, null);
}
```

Repeat this process for the rest of the Java files making sure to replace the highlighted portion above (*dashboard_fragment*) with the appropriate layout file. Note the use of R.layout - this specifies we are looking for a layout resource (recall that R. refers to resources of the specified type - we previously used R.id to find Views by their ID).
These previous steps handle our Fragment XML files, and create the corresponding Java class to use them as fragments. Note the use of LayoutInflater. When we call its inflate method, you can imagine inflating the XML in our activity UI. Specifically, this instantiates the specified XML code into the View just created. In this case, upon creating the specified Fragment, it will load

the corresponding XML code (i.e., from the specified XML layout file) and present it in the FrameLayout.

Next, we will add the necessary code to access and activate each Fragment inside our MainActivity.java. The first thing we need is a Fragment Manager; this lets you create Fragments, remove Fragment, or replace running Fragments with different ones. You can use the getSupportFragmentManager method to access a Fragment Manager. In your MainActivity.java (after onCreate), add a new method called *loadFragment* which returns a boolean and takes a fragment as a parameter.

```
private boolean loadFragment(Fragment frag)
{
    if(frag != null)
    {
        getSupportFragmentManager().beginTransaction().replace(R.id.fragment_container, frag).commit();
        return true;
    }
    else return false;
}
```

Note the somewhat complex syntax. The key method in the middle is *.replace* - as called, this replaces the FrameLayout we previously defined (called fragment_container) with the specified Fragment. Replace is one of several possible FragmentTransactions. The .commit method causes the replacement to actually happen when the main thread can schedule it.

As an example of how this function will be used, go to the onCreate method and add the following to the end of the method:

```
loadFragment(new HomeFragment());
```

As we know, onCreate is our app entry point; this call to loadFragment ensures that the application starts on the HomeFragment. You can run the app at this point to see this behaviour. The app should start on the HomeFragment with the text “Home” visible.

Now we will add functionality to our BottomNavigationView listener so that the Frame Layout will always load the selected screen via its Fragment. To do this involves creating a new Fragment of the same type as the navigation option selected, then returning our loadFragment function with the new fragment, as seen to the right:

```
private BottomNavigationView.OnNavigationItemSelectedListener
        = (item) → {

        Fragment fragment = null;

        switch (item.getItemId()) {
            case R.id.navigation_home:
                fragment = new HomeFragment();
                break;
            case R.id.navigation_dashboard:
                fragment = new DashboardFragment();
                break;
            case R.id.navigation_notifications:
                fragment = new NotificationsFragment();
                break;
            case R.id.navigation_settings:
                fragment = new SettingsFragment();
                break;
        }
        return loadFragment(fragment);
};
```

At this point, run your app. Try clicking on each navigation option on the BottomNavigationView. Make sure that the fragments change when you select new options on the navigation bar.

3. Functionality and Data Retention with ViewModel

You have now successfully created the navigation part of your app. We will now add functionality to the counters in each Fragment, and introduce the ViewModel class. The ViewModel class can be used to save and load data when the screen changes, effectively passing data between Fragments.

We will first add counter functionality to our HomeFragment class. To do this, we will need to create a reference to the Button and the TextView contained in the home_fragment layout. You will also need to create an integer to keep count. Lastly, we also need to store our View variable to make use of it when assigning the references before returning the inflated view. We will also implement the onClick method inside the onCreateView for easy access to the increment button:

```
public class HomeFragment extends Fragment {

    TextView counter;
    Button increment;
    int count = 0;
    View view;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        view = inflater.inflate(R.layout.home_fragment, null);
        counter = view.findViewById(R.id.home_counter);
        increment = view.findViewById(R.id.home_button);
        increment.setOnClickListener(new View.OnClickListener(){
            @Override
            public void onClick(View view){
                count++;
                counter.setText(Integer.toString(count));
            }
        });
        return view;
    }
}
```

A couple things to note here:

- We are now using inflater (the LayoutInflater) to get the home_fragment layout, and storing a reference to it in *view* (rather than immediately returning it as before)
- This is because we need to get the home_counter and home_button associated with the home_fragment layout, prior to returning the inflated layout. This is done using view.findViewById – similar to usual, but in this case, calls findViewById on Views specifically contained with the specified view

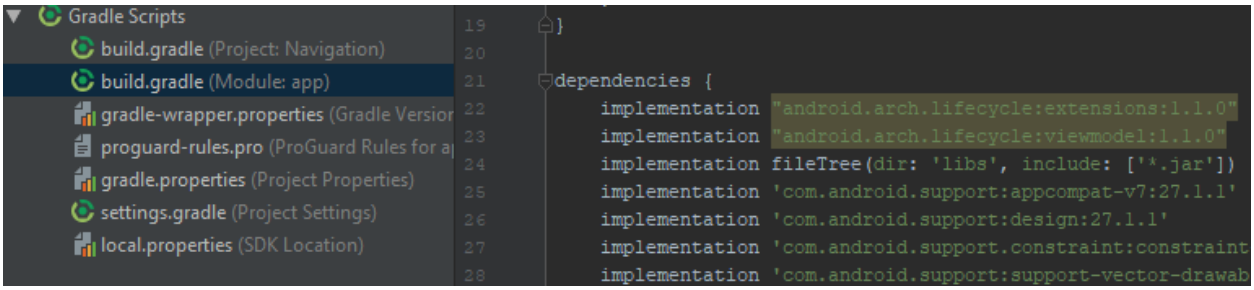
The process of setting up the onClickListener for the button should be familiar by now (as is the process of making it increment the counter).

Repeat this process for the other 3 fragments. Of course, you must modify the R.layout and R.id references for each. For example, instead of using R.layout.home_fragment, use R.layout.settings_fragment, and so on, for each Fragment.

Run your app, and cycle through your Fragments. Make sure that when you press the button on each screen, the TextView is incremented by one. Note that when you switch back to a previously changed counter, the counter resets. We can fix this with the ViewModel class which can store variables on Fragment changes.

To be able to use the ViewModel class, we have to add a couple of dependencies to our build.gradle file. Open the build.gradle file and add the following two dependencies:

```
implementation "android.arch.lifecycle:extensions:1.1.0"
implementation "android.arch.lifecycle:viewmodel:1.1.0"
```

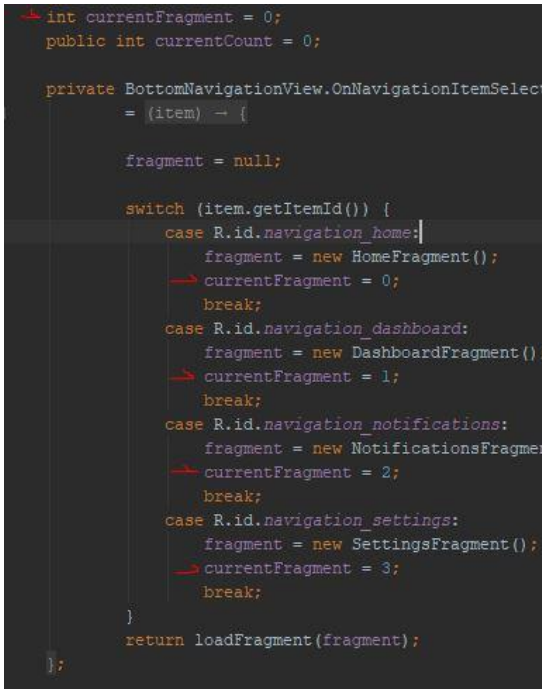


Note that you will need to resync your gradle.build file after doing this (you will be prompted above the text window).

After adding these dependencies, create a new java class called DataContainer. By now you have added classes several times to your project, so these instructions are not duplicated here. If you can't remember, scroll back through this Tutorial, or ask for help.

This class should extend ViewModel. For this class, simply add 4 integer variables, one for each of our fragments. Call them homeCounter, dashboardCounter, settingsCounter, and notificationsCounter. These will be used to get copies of the count from each Fragment as they are updated. This facilitates recovering the value when the screen changes, or even sharing data between fragments!

We will now go back to our MainActivity and add two ints to the class. Call them *currentFragment* and *currentCount*. We will assign a specific numeric value for each fragment. Home will be 0, Dashboard will be 1, Notifications will be 2, and Settings will be 3. Now, we must update the code in the *switch* statement so that any time we switch fragments, we will also update the value of *currentFragment* to the corresponding number, to keep track of which is active. See the image to the right:

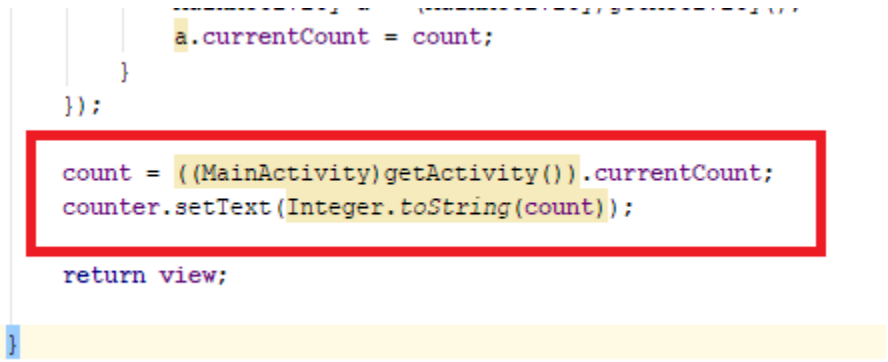


These values will help shortly when identifying what count variable to update in our *DataContainer* class. Since we start our activity in HomeFragment, go to HomeFragment.java to set up communication with the MainActivity. We need to pass values *through* the MainActivity, since it "houses" all the fragments, and the DataContainer.

We can get a reference to our MainActivity inside a given Fragment by using the method *getActivity()*. We will cast our MainActivity as shown in the image below (note the part inside the red box). You first get a reference to the Activity which houses the Fragment via getActivity. The result is cast to a MainActivity (called 'a' here), since getActivity returns a standard Activity (i.e., MainActivity extends Activity – recall OOP inheritance). Finally, we access the MainActivity's currentCount variable with a.currentCount. Note that this is all inside the onClick method. We first increase the counter (in the usual way), and then we update the currentCount in our MainActivity:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {

    view = inflater.inflate(R.layout.home_fragment,  root: null);
    counter = view.findViewById(R.id.home_counter);
    inc = view.findViewById(R.id.home_button);

    inc.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            count++;
            counter.setText(Integer.toString(count));

            MainActivity a = (MainActivity)getActivity();
            a.currentCount = count;
        }
```

This above code will update the currentCount variable in the MainActivity when the button is clicked. Let's add a couple more lines of code to also add the reciprocal bahaviour: to let the MainActivity update the Fragment's counter, when the Fragment is created. If we don't do this, the Fragment's own counter will be reset to 0 every time is it re-opened, since the Fragment is actually re-created each time it is reopened. Note that this would further result in our MainActivity.currentCount being reset each time the Fragment is opened. Right after and outside the onClick listener, add the following lines (in the red box):



Recall that this code is *all* inside the onCreateView method, so is executed every time the Fragment opens (which happens every time we change Fragments). These two new lines (in a red box) will again access the MainActivity, grab the value of currentCount, and update the count to this value. Next, we will update our count variables within the dataContainer class. But, before we get to the ViewModel part, repeat the above steps process for the other 3 fragments. No code changes are needed for the 4 lines added in this step, simple copy paste will do.

Try your app at this point. Pressing the button should increment the counter, as usual. Changing fragments will result in the same counter value being copied to the newly opened Fragment. The reason for this is that we have so far only copied the counter variable from each Fragment to update the currentCount in the MainActivity. In other words, all Fragments are incrementing the same counter! This is not what we want, so it's time to access our DataContainer (ViewModel) to rectify this. Review the ViewModel reference for further details of how they work, and what they're used for.

We will start by going into our MainActivity and declaring a DataContainer:

```
final DataContainer dc = ViewModelProviders.of(this).get(DataContainer.class);
```

Note the somewhat odd syntax. The final keyword ensures that this *reference* is unique and cannot be modified. The ViewModelProviders is a "factory" class that provides a ViewModel. The .of method ensures the ViewModel provided is associated with (i.e., in scope of) the current Activity. Finally, the .get method creates an instance of the provided class (in this case, our DataContainer) as the ViewModel. The end result is a special DataContainer object (called dc in

this case) that survives changing Fragments, and thus can contain all of our Fragment data we want to retain or pass between them.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    BottomNavigationView navigation = (BottomNavigationView) findViewById(R.id.navigation);
    navigation.setOnNavigationItemSelectedListener(mOnNavigationItemSelectedListener);

    final DataContainer dc = ViewModelProviders.of(this).get(DataContainer.class);

    loadFragment(new HomeFragment());
}

}
```

Now go to the loadFragment method you created earlier. We must modify it to get information from the DataContainer to update our fragments with the previously stored count. This is where we will use the currentFragment integer created earlier:

```
private boolean loadFragment(Fragment fragment){
  if(fragment != null){

    getSupportFragmentManager()
        .beginTransaction()
        .replace(R.id.fragment_container, fragment)
        .commit();

    if(currentFragment == 0){
      currentCount = ViewModelProviders.of(this).get(DataContainer.class).homeCounter;
    }else if(currentFragment == 1){
      currentCount = ViewModelProviders.of(this).get(DataContainer.class).dashboardCounter;
    }else if(currentFragment == 2){
      currentCount = ViewModelProviders.of(this).get(DataContainer.class).notificationsCounter;
    }else if(currentFragment == 3){
      currentCount = ViewModelProviders.of(this).get(DataContainer.class).settingsCounter;
    }
    return true;
  }else return false;
}
```

Finally, we will do the similar process to save the currentCount into the appropriate integer in our DataContainer. We will do this in the OnNavigationItemSelected method, but *before* actually changing the fragment, like so:

```
private BottomNavigationView.OnNavigationItemSelectedListener mOnNavigationItemSelectedListener
        = (item) -> {

    if(currentFragment == 0){
        ViewModelProviders.of(MainActivity.this).get(DataContainer.class).homeCounter = currentCount;
    }else if(currentFragment == 1){
        ViewModelProviders.of(MainActivity.this).get(DataContainer.class).dashboardCounter= currentCount;
    }else if(currentFragment == 2){
        ViewModelProviders.of(MainActivity.this).get(DataContainer.class).notificationsCounter= currentCount;
    }else if(currentFragment == 3){
        ViewModelProviders.of(MainActivity.this).get(DataContainer.class).settingsCounter= currentCount;
    }

    fragment = null;

    switch (item.getItemId()) {
```

Now run your app and check that the counters are indeed being saved. Increase the counter on the different fragments, and going back to check that the data is retained. This may seem small, but it is the foundation of inter-fragment communication. In other words, the ability to pass data between fragments allows you to, for example, change settings on one screen, and have these setting changes affect a different screen (much like “real” apps).

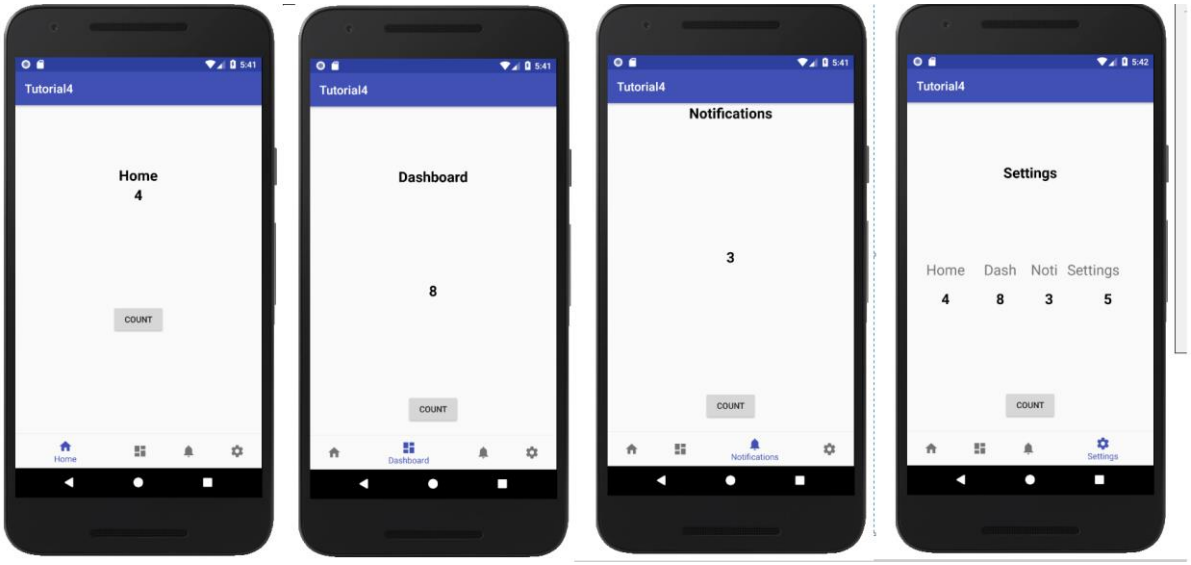
HomeWork

1. The way that we handled saving and getting data to our view model is by accessing variables in the MainActivity from the fragment and then letting the MainActivity handle the saving and loading. However, we can do this in a more direct way by allowing the fragments to access the ViewModel. This can be handy for inter-fragment communication (e.g., retaining settings from one fragment that affect another). This can be easily done by accessing the ViewModel from the fragment in the same way we did in the main activity, only difference is changing the context, like so:

   ```
   int count = ViewModelProviders.of((MainActivity)getActivity())
                                 .get(DataContainer.class).notificationCounter;
   ```

   Modify your current app implementation so that ***only*** the Fragments talk to the ViewModel in the fashion handled above. In other words, there should be no references to ViewModelProviders in the MainActivity.java file. Conversely, there should be no references to MainActivity.currentCount *anywhere*. You can delete currentCount from MainActivity, then remove the lines that use it throughout your code (you will no longer need it - just directly access the appropriate counters in the DataContainer). This way, MainActivity is only used to handle the fragment changing, and all fragments directly update their respective counter in the ViewModel.

2. Update the Settings Fragment to show the values in the counter for ALL of the other Fragments. In the settings_fragment layout file, add three more TextViews for counters. Duplicate the visual style of the current counter on the Fragment. Lay these and the previous counter TextView out nicely on the form using one of the Layouts we've seen in previous tutorials. Include a label beside each counter of the new TextViews, one for "Home", one for "Dashboard", one for "Notifications", and add a label as well for the "Settings" counter (careful to keep track of which is which with good naming conventions). In the SettingsFragment.java file, add code to access the ViewModel (as above) to get the corresponding count from each of the other Fragments. Overall, the behaviour should look like the following:



Optional Item - For Extra Practice and Brownie Points (but not actual points...):

3. Try changing Activities instead of Fragments. We worked through implementing the navigation bar and changing the screen through fragments. Start a new project and implement changing the screen by changing the main activity without using fragments. Similar to how it is done here. Add some sort of data to both activities that the user should input (a counter, some input text…), and following how to save and retrieve android instances, implement saving and retrieving your data for both of the activities.