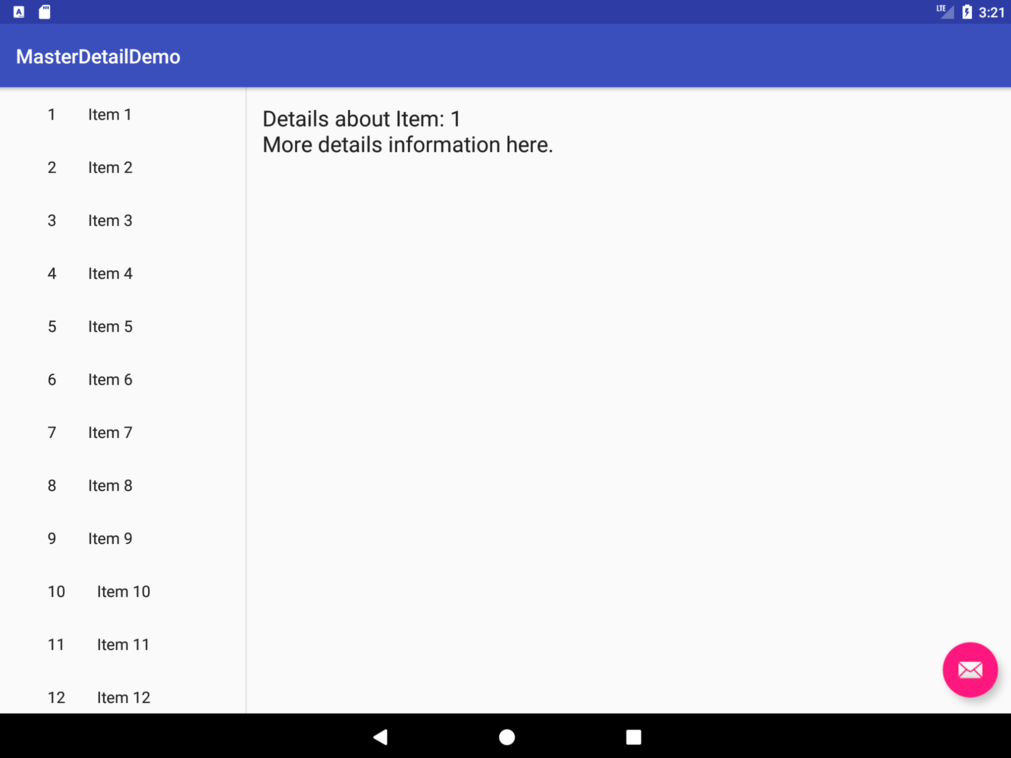
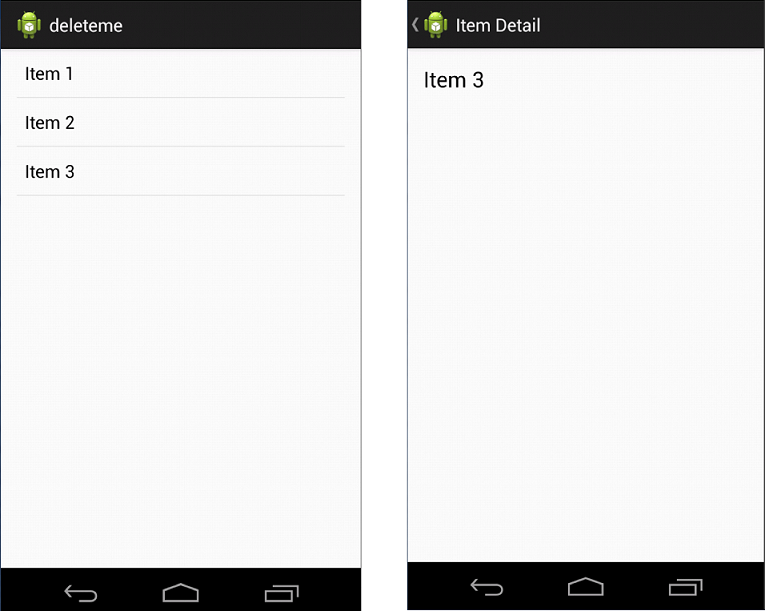
MASTER/DETAIL FLOW: (Refer App-MasterFlow)

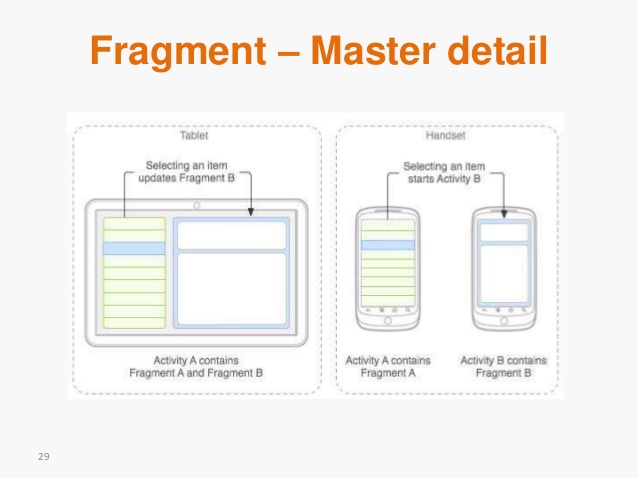
Here we shall learn about the concepts of a new UI layout called Master-Detail Flow. The layout is slightly different in case of Tablets and in case of phones. In case of phones, we shall have a Main Menu with certain list items and upon clicking the list items, a new activity will be opened which shall expand on the selected activity. In case of a tablet however, the Main Menu shall be displayed along a side pane which occupies around 1/4th of the screen real estate of the tablet. The remaining 3/4th is the expansion of the selected list item.



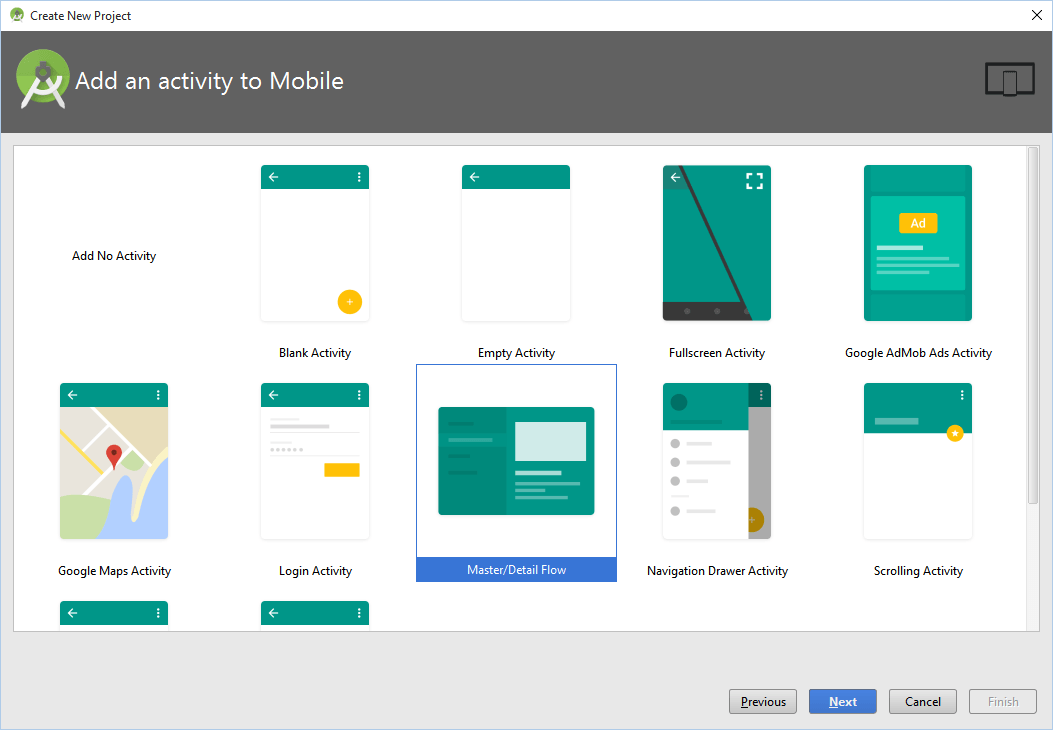
ON TABLETS



ON PHONES



In order to create a Master/Detail Flow activity, we select the corresponding option in the activity creator menu while creating a new project.



This shall add several Java Classes and several xml files to your project. The ones which we are interested in are:

1. DummyContent.java
2. ItemDetailFragment.java
3. ItemDetail.xml
4. AndroidManiFest.xml

Several of the functions in these classes have already been defined, we need to just tweak the predefined methods according to our usage. We here are creating an app which shall present the user with a list of different websites and the user-on selecting one of the items from the list-shall be shown the website in the same app inside a webviewer.

DUMMYCONTENT.JAVA

Upon scrolling to the end of the java file, we find that we have a class called DummyItem. This class gives the description or the template of all the items which we shall be adding in the Master List. Upon selecting the item from the Master List, we shall open the Detail View of the item showing the actual website.

By default this class has certain predefined attributes like ‘id’ and ‘content’. We here modify them to id,item\_name and url. The id will show the serial number of the item (website), the item name will show the name of the website, and the url will have the actual url of the website.

*/\*\*  
 \* A dummy item representing a piece of content.  
 \*/* **public static class** DummyItem {  
 **public final** String **id**;  
 **public final** String **item\_name**;  
 **public final** String **url**;  
  
 **public** DummyItem(String id, String item\_name, String url) {  
 **this**.**id** = id;  
 **this**.**item\_name** = item\_name;  
 **this**.**url** = url;  
 }  
  
 @Override  
 **public** String toString() {  
 **return item\_name**;  
 }  
 }  
}

Upon scrolling up, we see that there is a predefined HashMap called ITEM\_MAP with a for loop inside it. This is used to populate the list with some Dummy Items.

**public static final** List<DummyItem> ***ITEMS*** = **new** ArrayList<DummyItem>();  
  
*/\*\*  
 \* A map of sample (dummy) items, by ID.  
 \*/***public static final** Map<String, DummyItem> ***ITEM\_MAP*** = **new** HashMap<String, DummyItem>();  
  
**private static final int *COUNT*** = 25;  
  
**static** {  
 *// Add some sample items.* **for** (**int** i = 1; i <= ***COUNT***; i++) {  
 *addItem*(*createDummyItem*(i));  
  
 }

**private static void** addItem(DummyItem item) {  
 ***ITEMS***.add(item);  
 ***ITEM\_MAP***.put(item.**id**, item);

}

We need to add our own items now to this list, so we delete the for loop (or comment it) and then we manually add our own items. The final HashMap will look somewhat like:

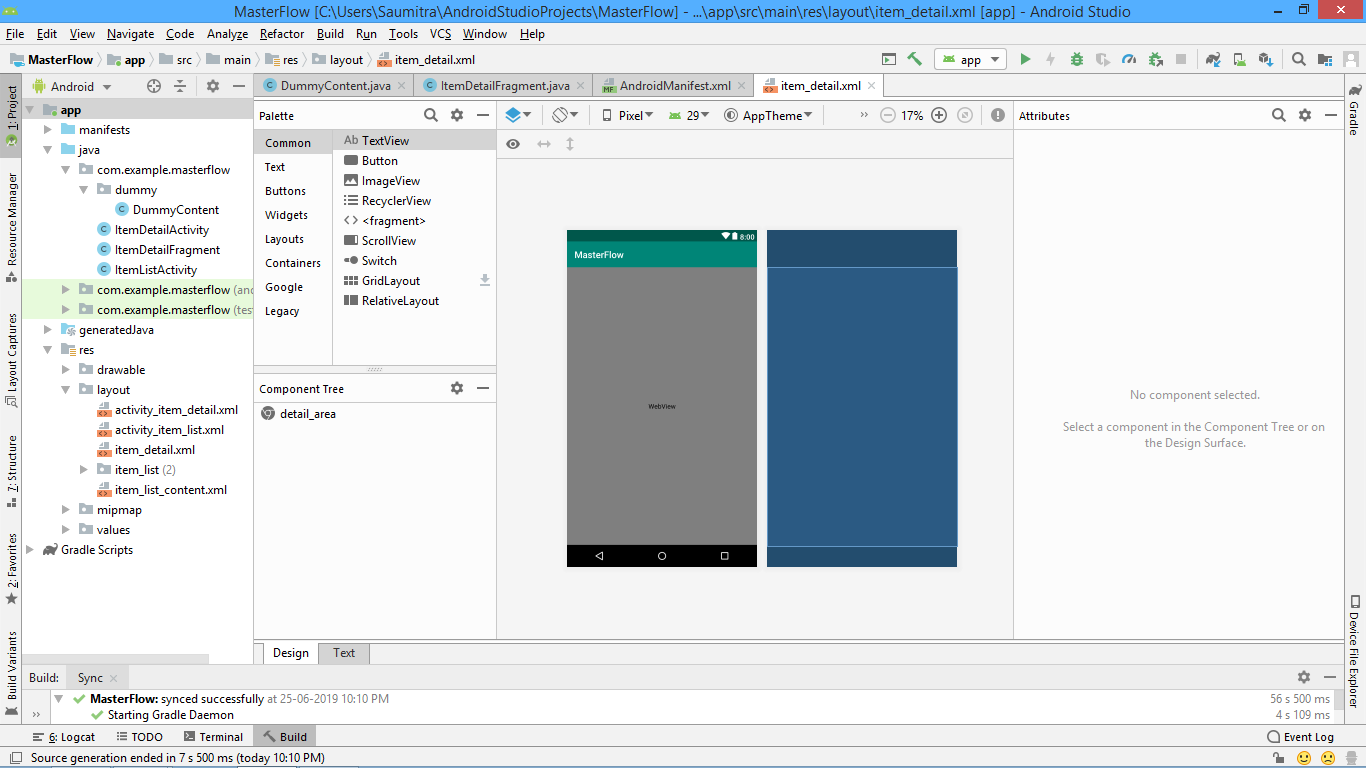
**public static final** Map<String, DummyItem> ***ITEM\_MAP*** = **new** HashMap<String, DummyItem>();  
  
**private static final int *COUNT*** = 25;  
  
**static** {  
 *// Add some sample items.  
 /\*for (int i = 1; i <= COUNT; i++) {  
 addItem(createDummyItem(i));  
  
 }  
 \*/  
 addItem*(**new** DummyItem(**"1"**,**"Google"**,**"https://www.google.com/"**));  
 *addItem*(**new** DummyItem(**"2"**,**"Is this Device On"**,**"http://www.ismycomputeron.com/"**));  
 *addItem*(**new** DummyItem(**"3"**,**"Heeeey"**,**"https://heeeeeeeey.com/"**));  
}  
  
**private static void** addItem(DummyItem item) {  
 ***ITEMS***.add(item);  
 ***ITEM\_MAP***.put(item.**id**, item);  
}

Now that we are done designing our list, we now move on to design the UI of the Detail Page.

ITEM\_DETAIL.XML:

By default this file only contains a textview which displays nothing. We first delete this TextView as we need to replace it with a webview. A webview is nothing but a dedicated space for displaying web content. We can resize and style the webview however we want. For now, let us limit our webview to the entire page. So, we set the layout\_height=match\_parent and layout\_width=match\_parent. We also assign an id to it as detail\_area.

*<?***xml version="1.0" encoding="utf-8"***?>*<**WebView xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_height="match\_parent"  
 android:layout\_width="match\_parent"  
 android:id="@+id/detail\_area"** />



ITEMDETAILFRAGMENT.JAVA

Now we come to the brains of the UI which we designed previously in the xml file. We first need to import the webviewer package in order to treat the webviewer as an object.

**import** android.webkit.WebView;

We finally scroll to the bottom of the java file where we can see the default implementation of the TextView in the onCreateView method which was previously present before we replaced it with a WebView. If we observe the upper portion of the method, then we shall see that the method has a view called rootView created which stores the inflated information of the item\_detail.xml file (the file we just modified). Now we need to create an object of the WebView and load the address stored in the url field of the list item. Thus, we use the loadUrl method of the WebView class. These are the only changes we make:

((WebView)rootView.findViewById(R.id.***detail\_area***)).loadUrl(**mItem**.**url**);

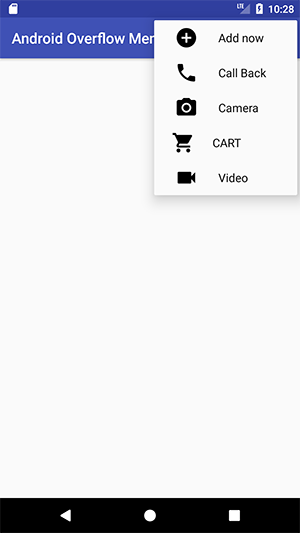
The entire method looks like:

@Override  
**public** View onCreateView(LayoutInflater inflater, ViewGroup container,  
 Bundle savedInstanceState) {  
 View rootView = inflater.inflate(R.layout.***item\_detail***, container, **false**);  
  
 *// Show the dummy content as text in a TextView.* **if** (**mItem** != **null**) {  
 ((WebView)rootView.findViewById(R.id.***detail\_area***)).loadUrl(**mItem**.**url**);  
 }  
  
 **return** rootView;  
}

SIDEBAR: We also need to replace all the field named as ‘content’ with ‘item\_name’. This is because we manually have defined the ‘item\_name’ attribute by replacing the ‘content’ attribute. Thus, wherever the word ‘content’ occurs, we replace it with ‘item\_name’.

OVERFLOW MENU (Refer App: OverflowMenu)

Here we shall learn how to handle the Overflow Menu. An overflow menu is nothing but the menu that pops up after selecting the 3 dots in the top right hand side corner.

We shall be designing an app where we create an overflow menu with 3 options denoting colors. Whenever the user selects one of these 3 options, the background color of the app will change to that color. In order to do so, only one of the options should be selected and not all.

In order to create such an app, we create a new project with a ‘Basic Activity’ and NOT AN ‘EMPTY ACTIVITY’. This by default includes the functionality for an overflow menu and we just need to manipulate it. Let us go through each of the files one by one:

MENU\_MAIN.XML

This file can be found under:

OverflowMenu\app\src\main\res\menu\menu\_main.xml

This shall by default contain an item called ‘Settings’. Now, because we need to customise the menu according to our needs, we delete the item.

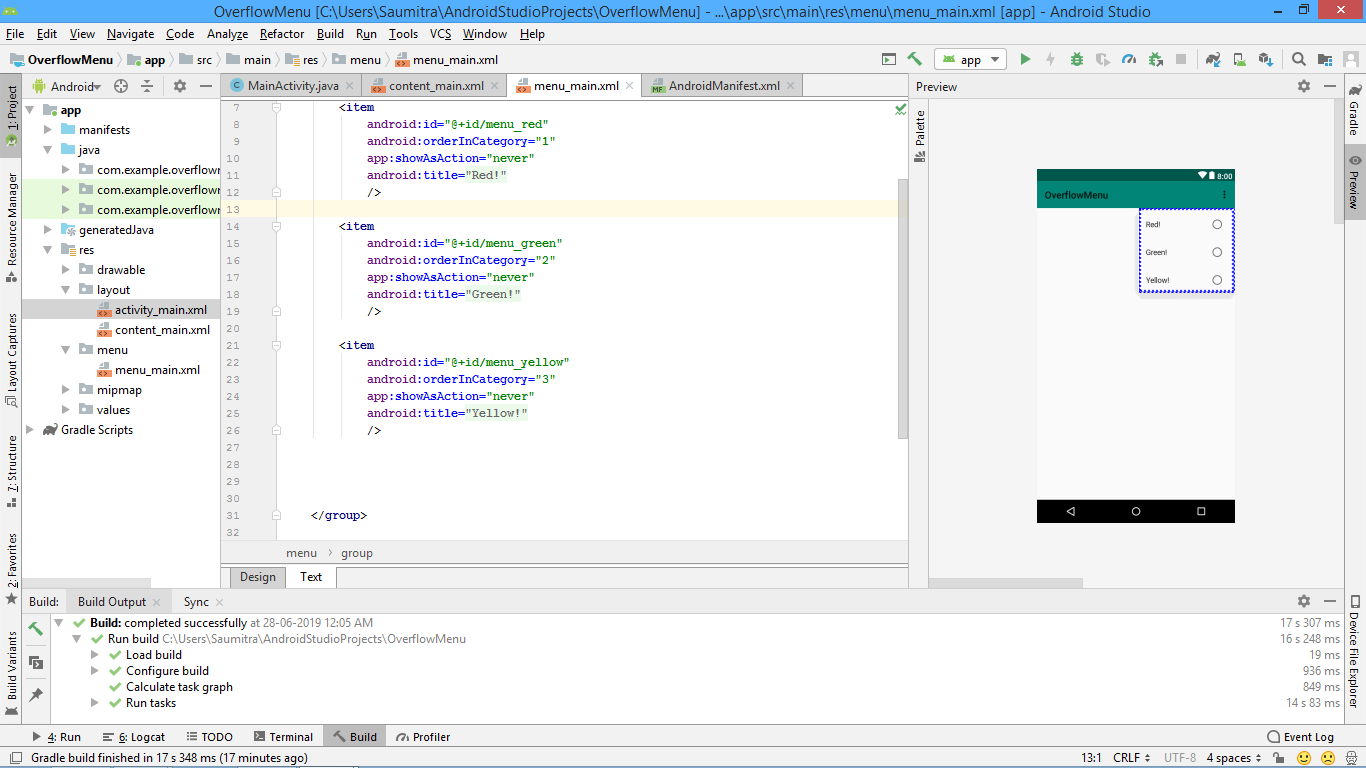
We now create a group of items. This helps in segregating the items. We shall use an attribute of the group tag called: ‘CheckableBehavior’ and set its value to ‘Single’. This means that at a given point of time ONLY ONE of the items in the group can be checked.

We then add items to the overflow menu using the ‘Item tag’. We also add various other attributes like:

1. Id: We set an id to each item thus enabling us to be able to refer to it as java objects.
2. orderInCategory: This allows us to set the element to be in a particular order in the overflow menu. This can be equal to ‘1’,’2’,’3’,etc.
3. Title: The name of the item to be shown in the OverflowMenu.
4. showAsAction: Currently we set it to ‘never’. But if required, (for quick actions) we can also manipulate its values to always or ‘ifRoomAvailable’. Doing so will set the element in a quick action tab for easy access.

The final xml file looks like:

<**group android:checkableBehavior="single"**>  
 <**item  
 android:id="@+id/menu\_red"  
 android:orderInCategory="1"  
 app:showAsAction="never"  
 android:title="@string/red\_string"** />  
  
 <**item  
 android:id="@+id/menu\_green"  
 android:orderInCategory="2"  
 app:showAsAction="never"  
 android:title="@string/green\_string"** />  
  
 <**item  
 android:id="@+id/menu\_yellow"  
 android:orderInCategory="3"  
 app:showAsAction="never"  
 android:title="@string/yellow\_string"** />  
  
</**group**>



MAIN ACTIVITY.JAVA

In the java file we shall see that there are various functions already defined for the ‘SnackBar’ as well as the ‘OverflowMenu’. If we scroll to the bottom of the java file, we shall see that there is a function called onOptionsItemSelected.

This is the function which is responsible for executing actions on clicking the overflow menu. We first create an object of our ConstraintLayout (or RelativeLayout as the case may be) and set it equal to the id of our layout.

ConstraintLayout mylayout=findViewById(R.id.***main\_view***);

Because we have 3 elements in the overflow menu, we create a switch case block switching the id of the item that was clicked.The background color can be set using a simple method of mylayout called setBackgroundColor. The overall method looks like:

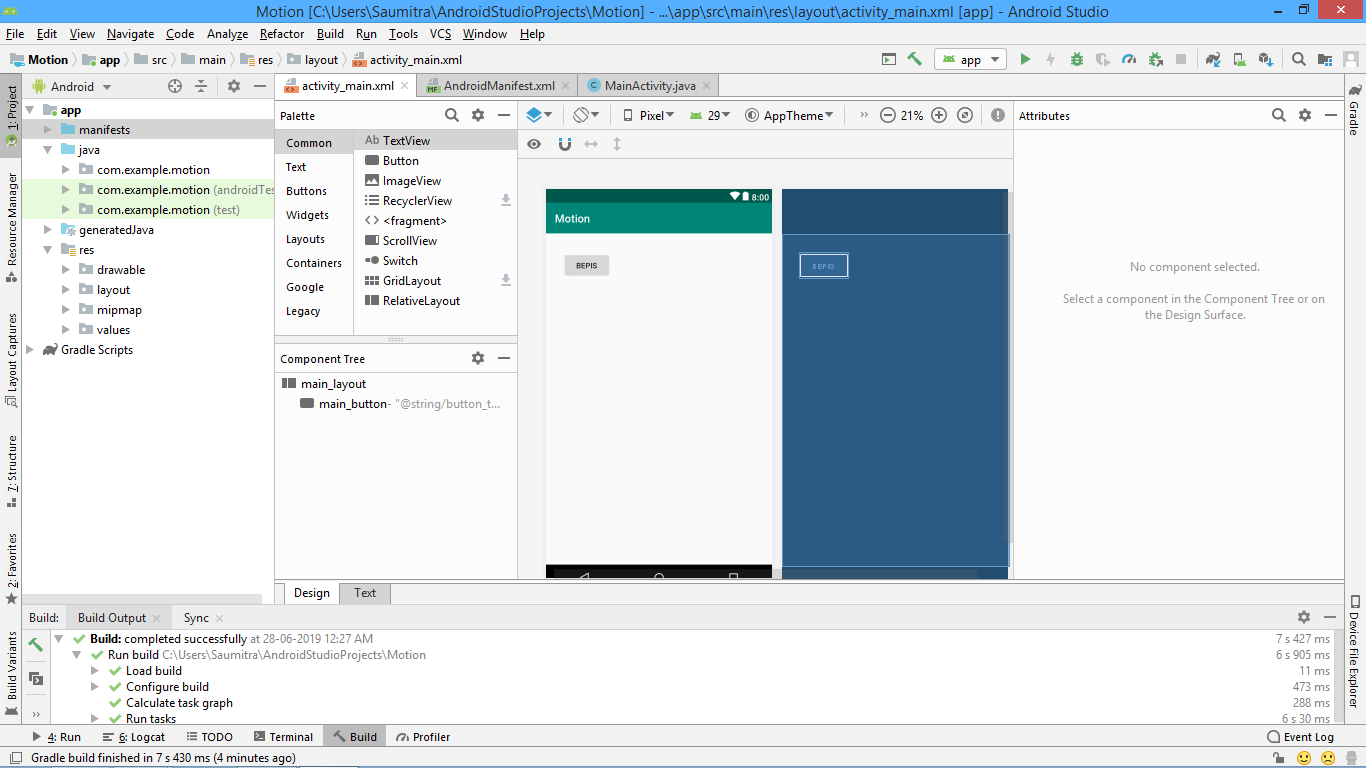
@Override  
 **public boolean** onOptionsItemSelected(MenuItem item) {

*// Handle action bar item clicks here. The action bar will  
 // automatically handle clicks on the Home/Up button, so long  
 // as you specify a parent activity in AndroidManifest.xml.* ConstraintLayout mylayout=findViewById(R.id.***main\_view***);  
 **switch** (item.getItemId())  
 {  
 **case**(R.id.***menu\_red***):  
 {  
  
 item.setChecked(**true**);  
 mylayout.setBackgroundColor(Color.***RED***);  
  
 **break**;  
 }  
  
 **case**(R.id.***menu\_green***):  
 {  
 item.setChecked(**true**);  
 mylayout.setBackgroundColor(Color.***GREEN***);  
 **break**;  
 }  
  
 **case**(R.id.***menu\_yellow***):  
 {  
 item.setChecked(**true**);  
 mylayout.setBackgroundColor(Color.***YELLOW***);  
 **break**;  
 }  
  
  
  
 }  
  
  
 **return super**.onOptionsItemSelected(item);  
 }  
}

TRANSITIONS AND ANIMATIONS: (Refer App: Motion)

In this app we shall learn the basics about transitions and animations. This app will simply listen for clicks on the screen and then smoothly transition a button from its initial position to the bottom right corner of the screen, while also increasing its size.

In order to do so, we simply first create a button and give it an id as well as some text on it (although the text does not matter). The final design screen looks like:



MainActivity.java

Here, we first import few important classes:

**import** android.view.View;  
**import** android.view.ViewGroup;  
**import** android.view.MotionEvent;  
**import** android.widget.RelativeLayout;  
**import** android.widget.Button;  
**import** android.transition.TransitionManager;

First we create an object of the view class which is used to refer to the RelativeLayout. We use findViewById in order to assign the RelativeLayout to the view.

Then we create an onTouchListener for the view. Inside the onTouchListener, we again create an object of the Button class and reference it to the button.

**myview**=findViewById(R.id.***main\_layout***);  
  
**myview**.setOnTouchListener(  
 **new** RelativeLayout.OnTouchListener() {  
 @Override  
 **public boolean** onTouch(View view, MotionEvent motionEvent) {  
  
 Button mybutton=findViewById(R.id.***main\_button***);

Then we change certain properties of the button like position and size:

1. Position:

In order to change the position, we first create a container called buttonPosition around the button using RelativeLayout.LayoutParams.

RelativeLayout.LayoutParams buttonPosition=**new** RelativeLayout.LayoutParams(  
 RelativeLayout.LayoutParams.***WRAP\_CONTENT***,RelativeLayout.LayoutParams.***WRAP\_CONTENT***);

This container is then given rules to position it to the bottom right corner of the screen.

buttonPosition.addRule(RelativeLayout.***ALIGN\_PARENT\_BOTTOM***,RelativeLayout.***TRUE***);  
buttonPosition.addRule(RelativeLayout.***ALIGN\_PARENT\_RIGHT***,RelativeLayout.***TRUE***);

Finally, we associate this container with the actual button that we had created.

mybutton.setLayoutParams(buttonPosition);

1. Size:

In order to change the size WHILE it is transitioning to the bottom of the screen, we create an object of the Viewgroup rather than RelativeLayout.LayoutParams, and set it equal to a method called getLayoutParams.

SIDEBAR: We can also create an object of RelativeLayout.LayoutParams, but we need to typecast the RHS to an object of RelativeLayout.LayoutParams

This method returns the current attributes of the box surrounding the button.

ViewGroup.LayoutParams buttonSize=mybutton.getLayoutParams();

We then adjust its height and width:

buttonSize.**width**=450;  
buttonSize.**height**=300;

Finally, we add these LayoutParams to the button.

mybutton.setLayoutParams(buttonSize);

1. Adding Animation:

In order to add a DEFAULT animation from all the initial conditions of the button to its absolute final state, we have a helper function of the TransitionManager class called beginDelayedTransition. It accepts the view in which the changes are taking place and transitions it automatically from the initial state to the final state.

FINAL JAVA CODE:

**package** com.example.motion;  
  
**import** androidx.appcompat.app.AppCompatActivity;  
  
**import** android.os.Bundle;  
**import** android.view.View;  
**import** android.view.ViewGroup;  
**import** android.view.MotionEvent;  
**import** android.widget.RelativeLayout;  
**import** android.widget.Button;  
**import** android.transition.TransitionManager;  
  
**public class** MainActivity **extends** AppCompatActivity {  
  
 ViewGroup **myview**;  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
  
 **myview**=findViewById(R.id.***main\_layout***);  
  
 **myview**.setOnTouchListener(  
 **new** RelativeLayout.OnTouchListener() {  
 @Override  
 **public boolean** onTouch(View view, MotionEvent motionEvent) {  
  
 Button mybutton=findViewById(R.id.***main\_button***);  
 TransitionManager.*beginDelayedTransition*(**myview**);  
  
 *//CHANGING THE POSITION OF THE BUTTON* RelativeLayout.LayoutParams buttonPosition=**new** RelativeLayout.LayoutParams(  
 RelativeLayout.LayoutParams.***WRAP\_CONTENT***,RelativeLayout.LayoutParams.***WRAP\_CONTENT*** );  
 buttonPosition.addRule(RelativeLayout.***ALIGN\_PARENT\_BOTTOM***,RelativeLayout.***TRUE***);  
 buttonPosition.addRule(RelativeLayout.***ALIGN\_PARENT\_RIGHT***,RelativeLayout.***TRUE***);  
  
 mybutton.setLayoutParams(buttonPosition);  
  
 *//CHANGING THE SIZE OF THE BUTTON  
 // ViewGroup.LayoutParams buttonSize=mybutton.getLayoutParams();* RelativeLayout.LayoutParams buttonSize= (RelativeLayout.LayoutParams) mybutton.getLayoutParams();  
 buttonSize.**width**=450;  
 buttonSize.**height**=300;  
 mybutton.setLayoutParams(buttonSize);  
  
  
 **return true**;  
 }  
 }  
 );  
  
 }  
}