CS4131
Mobile Application Development

Name: _____ ( ) Date: _____

Chapter 4: Android Action Bars; Scrollable and Tabbed Layouts

4.1 Introduction to Chapter 4

We currently have the means to build very simple, single-activity apps though the use of various ViewGroups, Views and their associated event-listeners and multi-touch event handling and gesture detection. Now, we wish to take it to the next level and add multiple functionalities on our app toolbars and bring in scrollable and tabbed features to our apps which you see in many apps today.

4.2 Action Bars: Overflow Menus and Action Buttons

The app bar, also known as the action bar, is one of the most important design elements in your app's activities, because it provides a visual structure and interactive elements that are familiar to users. Using the app bar makes your app consistent with other Android apps, allowing users to quickly understand how to operate your app and have a great experience. The key functions of the app bar are as follows:

- A dedicated space for giving your app an identity and indicating the user's location in the app.
- Access to important actions in a predictable way, such as search.
- Support for navigation and view switching (with tabs or drop-down lists).

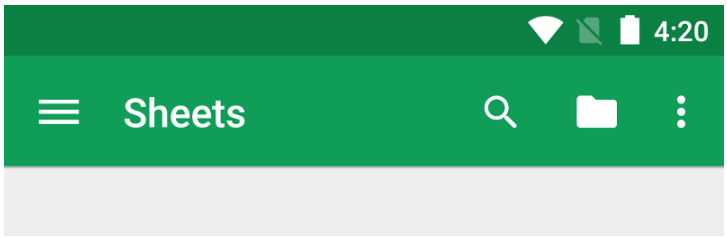


Figure 1: Example of Action Bar

An action bar has four main functional areas, namely the app icon, view control (dedicated space for app titles and option to switch between views), action buttons (mainly for important actions) an action overflow (unimportant actions shown in an overflow menu). The diagram on the right outlines the overview just described:

To set up the action bar to contain the functionalities described, a little bit of work will be required. An example app will be used to describe what is required.

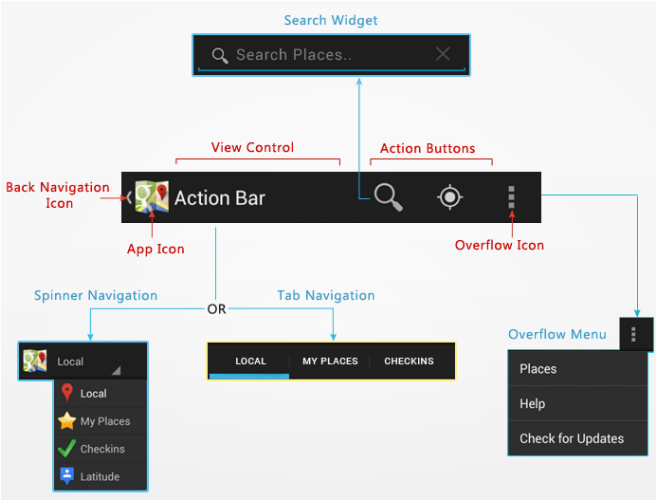


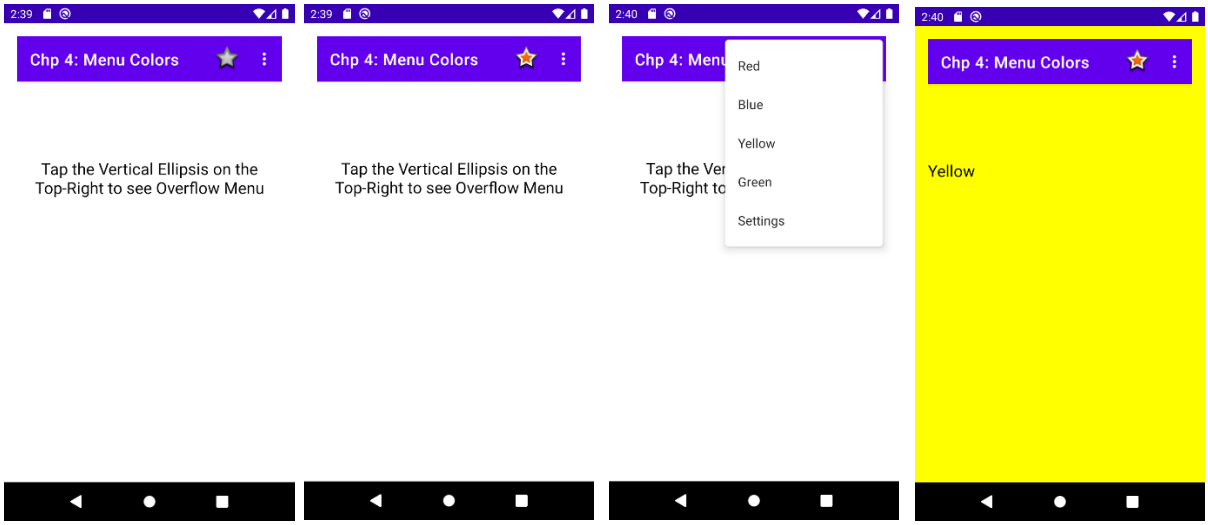
Figure 2: Overview of Action Bar

Example 1: MenuColors

First the theme will have to be edited to reflect some custom styles. Essentially, we are defining the styles for the action bar and other overlays from existing themes that are present in their libraries, while still retaining the app's existing colour schemes (or styles in this case).

The example app in this case allows the user to toggle the "favourite" icon which serves as an action icon, as well as allow the user to select menu items from the overflow menu, such that picking the colour will change the background to the corresponding colour option. Selecting "Settings" will reset the app background colour and TextView.

Screenshots of MenuColors app

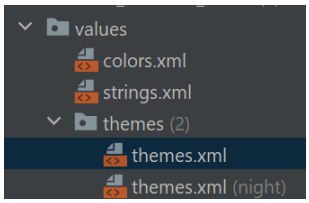


Open the themes.xml and add the following lines:

themes.xml

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.MenuColors" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor" tools:targetApi="l">?attr/colorPrimaryVariant</item>
        <!-- Customize your theme here. -->
    </style>

    <style name="Theme.MenuColors.NoActionBar">
        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>

    <style name="Theme.MenuColors.AppBarOverlay" parent="ThemeOverlay.AppCompat.Dark.ActionBar" />
    <style name="Theme.MenuColors.PopupOverlay" parent="ThemeOverlay.AppCompat.Light" />
</resources>
```

If you are also implementing a night-mode in your app, you may also want to edit the themes.xml labelled night. Essentially why there are two versions of themes.xml is because physically, if you explore your computer's File Explorer, there are two folders containing the themes.xml in your resources, one specifically for normal day mode and the other for night mode.

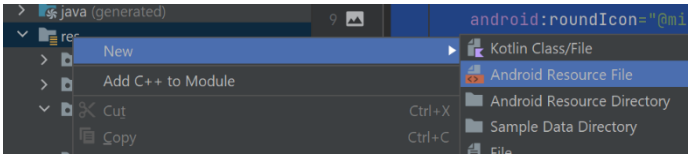


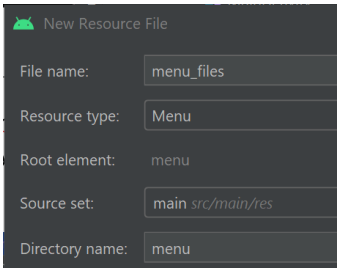
After that you will need to edit the AndroidManifest.xml to take reference to the theme:
AndroidManifest.xml

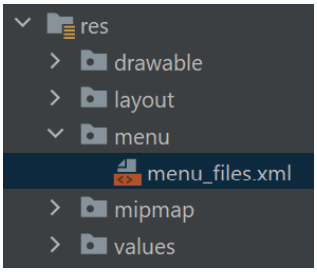
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.menucolors">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MenuColors">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:theme="@style/Theme.MenuColors.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The important part here is to add the NoActionBar theme into your activity. Note that MenuColors is the name of the project and hence, the string "Theme.MenuColors.NoActionBar". For your own personal projects, just replace MenuColors with your project's name.

To add an overflow menu and action buttons, you will first need to create a menu folder and a menu XML resource file. Right-click the res folder and add a new Android Resource File. Name the file "menu_files" and change the Resource Type to "Menu". The rest should be the defaults and click OK. You will see that your XML file is created under the menu folder of your resources, as seen on the right.







In the menu_files.xml, you can now add your options in your overflow menu, as well as any action buttons you wish to define. By app development conventions and some logical sense too, as the action bar has limited space, **any action buttons MUST be in in the form of icons** (strictly no words). Also, by **Material Design conventions**, which govern the conventions of good, meaningful app design, for the icon placement, **place most-used actions on the left**, progressing towards the **least-used actions on the far right**. **Any remaining actions that don't fit on the app bar can go into an overflow menu**.

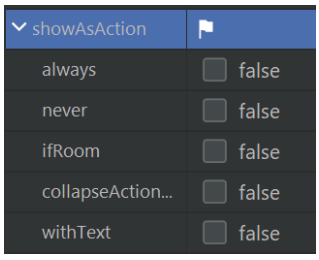
You may read more here: https://material.io/components/app-bars-top#anatomy

menu_files.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity">

    <group >
        <item
            android:id="@+id/menu_red"
            android:title="@string/menu_red" />
        <item
            android:id="@+id/menu_blue"
            android:title="@string/menu_blue" />
        <item
            android:id="@+id/menu_yellow"
            android:title="@string/menu_yellow" />
        <item
            android:id="@+id/menu_green"
            android:title="@string/menu_green" />
        <!--"Mark Favourite", should appear as action button if possible -->
        <item
            android:id="@+id/action_favourite"
            android:icon="@android:drawable/star_big_off"
            android:title="@string/action_favorite"
            app:showAsAction="ifRoom" />
        <!-- Settings, should always been in the overflow -->
        <item
            android:id="@+id/action_settings"
            android:orderInCategory="100"
            android:title="@string/action_settings"
            app:showAsAction="never" />
    </group>
</menu>
```

strings.xml

```
<resources>
    <string name="app_name">Chp 4: Menu Colors</string>
    <string name="initial">Tap the Vertical Ellipsis on the Top-Right to see Overflow Menu</string>
    <string name="action_favorite">Favourite</string>
    <string name="action_settings">Settings</string>
    <string name="menu_red">Red</string>
    <string name="menu_blue">Blue</string>
    <string name="menu_green">Green</string>
    <string name="menu_yellow">Yellow</string>
</resources>
```



As an explanation for the above, note that all menu items, regardless of whether they are in the overflow menu or the action bar, will be under one group. In general, you can define your menu items to be in different groups depending on your purpose. To define if a menu item is to be an action button, depends on the app:showAsAction attribute. The options of the attribute are detailed below:

showAsAction	Description
always	Always place this item in the app bar. Avoid using this unless it's critical that the item always appear in the action bar. Setting multiple items to always appear as action items can result in them overlapping with other UI in the app bar.

never	Never place this item in the app bar. Instead, list the item in the app bar's overflow menu.
ifRoom	Only place this item in the app bar if there is room for it. If there is not room for all the items marked "*ifRoom*", the items with the lowest **orderInCategory** values are displayed as actions, and the remaining items are displayed in the overflow menu. Note that orderInCategory is another attribute available in each menu item
collapseActionView	The action view associated with this action item (as declared by android:actionLayout or android:actionViewClass) is collapsible. Collapsible layouts will be discussed in a future section.
withText	Also include the title text (defined by android:title) with the action item. You can include this value along with one of the others as a flag set, by separating them with a pipe |.

Now we can define the main layout as well as the code:
activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/mainLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingTop="16dp"
    android:paddingRight="16dp"
    android:paddingBottom="16dp"
    tools:context=".MainActivity">
    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/Theme.MenuColors.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/my_toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            android:elevation="4dp"
            app:popupTheme="@style/Theme.MenuColors.PopupOverlay" />

    </com.google.android.material.appbar.AppBarLayout>
    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="150dp"
        android:text="@string/initial"
        android:textAlignment="center"
        android:textColor="@color/black"
        android:textSize="20sp" />
</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

In the above, we observe that we will need to define the **Toolbar** in the layout. The Toolbar will be the one holding all your action bar features, and will be called in your code to perform any action bar related functionalities. Notice that the Toolbar is defined inside the AppBarLayout. The AppBarLayout is a vertical LinearLayout which implements many of the features of material designs app bar concept, namely scrolling gestures. **It depends heavily on being used as a direct child within a CoordinatorLayout**, hence, using the AppBarLayout on other Layouts will not work as well. The AppBarLayout in this case is used to retain the style and themes we use on this app. In fact, it is also responsible for any scrolling behaviours which will be explored in a future section.

MainActivity.kt

```
1   import android.graphics.Color
2   import androidx.appcompat.app.AppCompatActivity
3   import android.os.Bundle
4   import android.view.Menu
5   import android.view.MenuItem
6   import android.widget.TextView
7   import androidx.appcompat.widget.Toolbar
8   import androidx.coordinatorlayout.widget.CoordinatorLayout
9
10  class MainActivity : AppCompatActivity() {
11      private var isFav = false
12
13      override fun onCreate(savedInstanceState: Bundle?) {
14          super.onCreate(savedInstanceState)
15          setContentView(R.layout.activity_main)
16          setSupportActionBar(findViewById(R.id.my_toolbar))
17      }
18      override fun onCreateOptionsMenu(menu: Menu?): Boolean {
19          menuInflater.inflate(R.menu.menu_files, menu)
20          return true
21      }
22      override fun onOptionsItemSelected(item: MenuItem): Boolean {
23          /* Handle action bar item clicks here. Action bar will auto handle clicks on
24          the Home/Up button, as long as you specify a parent activity in AndroidManifest.xml */
25          val mainLayout : CoordinatorLayout = findViewById(R.id.mainLayout)
26          val textView :TextView = findViewById(R.id.textView)
27          when (item.itemId){
28              R.id.menu_red -> { mainLayout.setBackgroundColor(Color.RED)
29                  textView.setText(R.string.menu_red)
30                  textView.setTextColor(Color.WHITE)
31                  return true
32              }
33              R.id.menu_blue -> { mainLayout.setBackgroundColor(Color.BLUE)
34                  textView.setText(R.string.menu_blue)
35                  textView.setTextColor(Color.WHITE)
36                  return true
37              }
38              R.id.menu_green -> { mainLayout.setBackgroundColor(Color.GREEN)
39                  textView.setText(R.string.menu_green)
40                  textView.setTextColor(Color.BLACK)
41                  return true
42              }
43              R.id.menu_yellow -> { mainLayout.setBackgroundColor(Color.YELLOW)
44                  textView.setText(R.string.menu_yellow)
45                  textView.setTextColor(Color.BLACK)
46                  return true
47              }
48              R.id.action_favourite -> {
49                  val iconStr :String
50                  if (isFav) {
```

```
51                      iconStr = "star_big_off"
52                      isFav = false
53                  } else {
54                      iconStr = "star_big_on"
55                      isFav = true
56                  }
57                  val imgID = resources.getIdentifier("android:drawable/$iconStr", null, null)
58                  item.setIcon(imgID)
59                  return true
60              }
61              R.id.action_settings -> {
62                  mainLayout.setBackgroundColor(Color.WHITE)
63                  textView.setText(R.string.initial)
64                  textView.setTextColor(Color.BLACK)
65                  return true
66              }
67              else -> return super.onOptionsItemSelected(item)
68          }
69      }
70  }
```

Explanation:

Lines 16 Lines 18 to 21	The SetSupportActionBar() function is used to setup the action bar from the resource toolbar. Note that the function call MUST be place AFTER the SetContentView() is called. You will also notice an overridden onCreateOptions() method. This is to setup the overflow menu and action icons with the appropriate menu resource file. Note that in general, an inflater call will be responsible for “inflating” a resource blueprint onto the app context (the device using the app)
Lines 22 to 69	The onOptionItemSelected() method is overridden which serves as the “listener” for any menu item picked (both the overflow menu and the action icons). In this case, you can use a switch statement (when…) to perform the respective actions when a certain menu item is clicked on.

4.3 Fragments

A fragment is a self-contained, modular section of an app’s user interface and corresponding behaviour that can be embedded within an activity. Fragments can be assembled to create an activity during the application design phase and added to or removed from an activity during the app’s runtime to create a dynamically changing user interface.

Fragments may only be used as part of an activity and cannot be instantiated as standalone app elements. A fragment can also be thought of as a functional “sub-activity” with its own lifecycle similar to that of a full activity. You may observe that the fragment life cycle is very similar to that of the activity life cycle you explored in Lab 1 and you may also observe the relationship between a fragment and its associated activity during the running of the app, both in Figure 3.

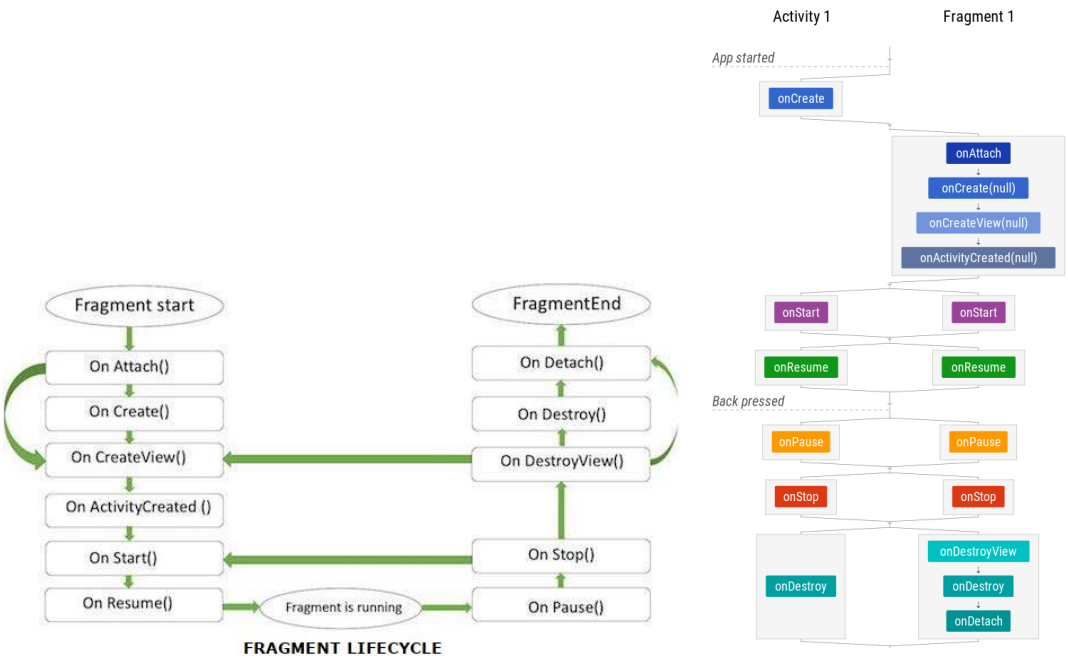


Figure 3: Fragment Life Cycle and Relationship with Activity

The key part here to notice is that fragments within an activity are created only AFTER the associated activity is created and there is a sequence at which the fragment must follow (onAttach, onCreate, onCreateView, onActivityCreated) for different parts of the life cycle. The same can be said likewise when the fragment or activity is destroyed, where while it appears they can be destroyed, the fragment needs to destroy and detach itself first before the activity is destroyed.

The advantage of using a fragment is that it can be seen as and used as an individual Android component which can be used as part of an activity's user interface, hence, encapsulating the specific functionality so that it can facilitate easy use and implementation. Fragments can provide for modularity and reusability of your UI components and code, allowing developers to easily create complex activity codes across fragments while making it easy for organization and maintenance. Also, one of the main difficulties of app development is to accommodate to a wide variety of screen-sizes, where fragments partially solve the problem, such that fragments can be used to represent parts of the UI, and depending on screen-size, it is now easy to programmatically define how the parts of the UI can be represented depending on screen size.

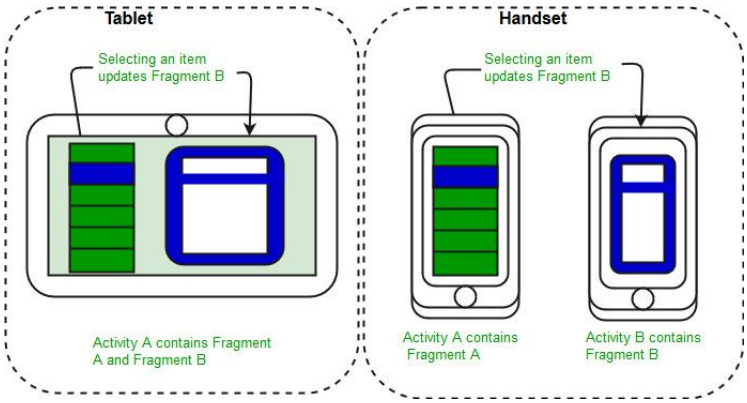
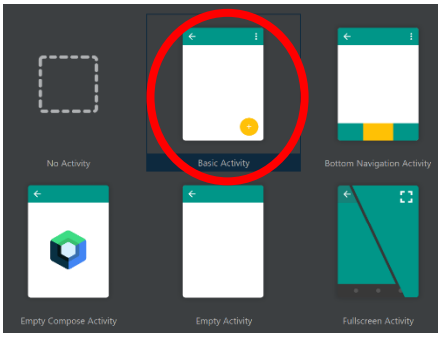
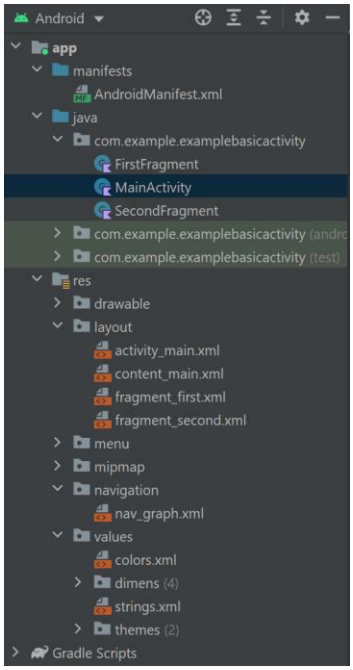


Figure 4: Fragment's Seen as Sections of a UI



To start utilizing fragments in your app, instead of selecting a new Empty Activity, you may select a new Basic Activity. After a new Basic Activity is selected, you will find that you now have many more resources defined, as well as two fragments and an activity, with their respective XML layouts, already setup for you. If you look at the fragment and activity codes, you will notice that they have, by default, set up a working overflow menu, a Floating Action Button (FAB) and basic navigation features (which we do not need for now).



For the apps in this chapter, we do not need certain components from the basic activity. You can do the following:

- Delete the entire navigation folder in res
- Delete the overridden onSupportNavigateUp() method in MainActivity.kt
- Delete the following lines in the overridden onCreate() method in MainActivity.kt

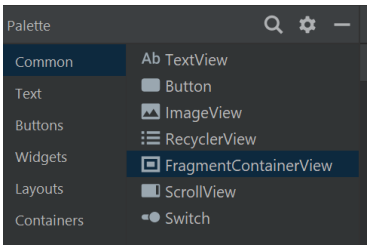
```
val navController = findNavController(R.id.nav_host_fragment_content_main)
appBarConfiguration = AppBarConfiguration(navController.graph)
setupActionBarWithNavController(navController, appBarConfiguration)
```

- Delete the overridden onViewCreated() method in BOTH FirstFragment.kt and SecondFragment.kt
- Delete the nav_host_fragment_content_main View object from content_main.xml

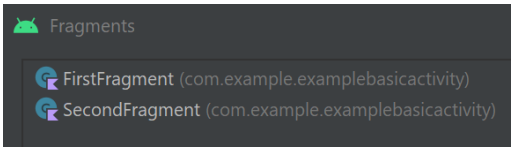
You may notice the code for MainActivity.kt, FirstFragment.kt and SecondFragment.kt have binding going on. Such a feature is known as ***View Binding*** and it View binding is a feature that allows you to more easily write code that interacts with views. Once view binding is enabled in a module, it generates a binding class for each XML layout file present in that module. An instance of a binding class contains direct references to all views that have an ID in the corresponding layout. You may read more in the following link: https://developer.android.com/topic/libraries/view-binding

Now your project is ready for use. We shall see the fragments applied in the following example.

Example 2: Fragments



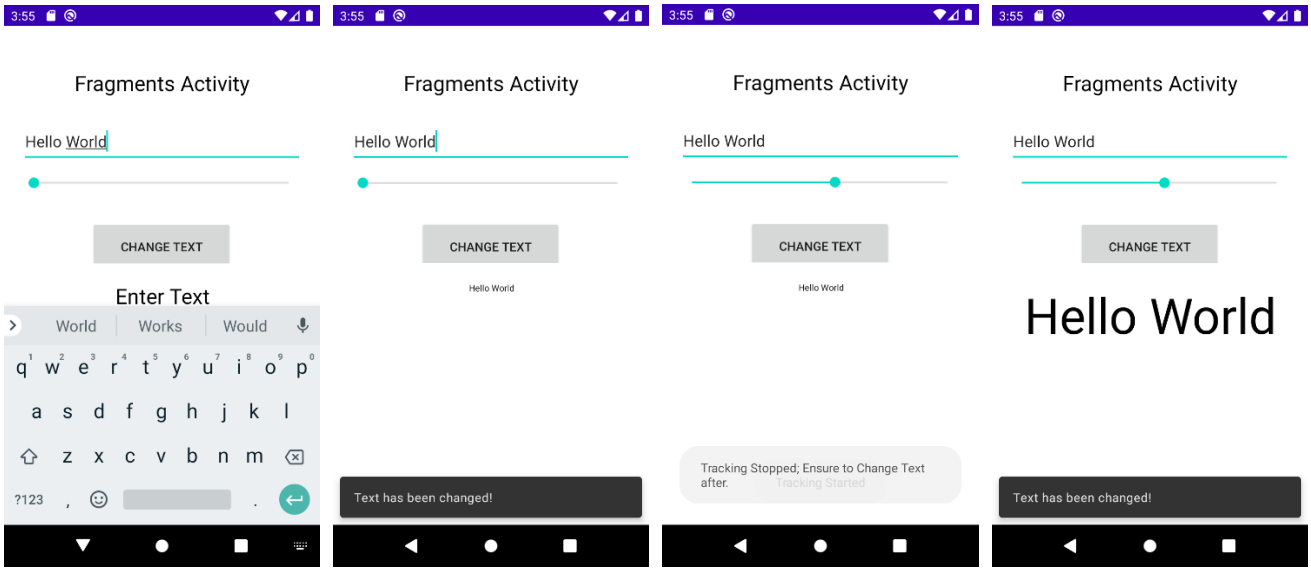
The app utilizes two fragments, one containing the EditText, SeekBar and Button, and the other only containing the TextView which showcases the output.



For this app, fragments are added into activity_main.xml. To do that, we use a FragmentContainerView to contain an individual fragment. The FragmentContainerView is easily accessed through the design mode palette and when pulled into the layout, you are given the option to choose which fragment should the container contain.

Note for this example, the fragments' Kotlin file and associated XML files were renamed.

Screenshots of Fragments app



strings.xml

```
<resources>
    <string name="app_name">Chp 4: Fragments</string>
    <string name="mainText">Fragments Activity</string>
    <string name="EditTextPrompt">Enter Text</string>
    <string name="BtnText">Change Text</string>
    <string name="BtnClick">Text has been changed!</string>
    <string name="seekStart">Tracking Started</string>
    <string name="seekStop">Tracking Stopped; Ensure to "Change Text" after.</string>
</resources>
```

TextFragment.kt

```
1   import android.os.Bundle
2   import androidx.fragment.app.Fragment
3   import android.view.LayoutInflater
4   import android.view.View
5   import android.view.ViewGroup
6   import android.widget.TextView
7
8   class TextFragment : Fragment() {
9       private lateinit var textview : TextView
10
11      override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
12          savedInstanceState: Bundle?): View? {
13          // Inflate the layout for this fragment
14          val view : View = inflater.inflate(R.layout.fragment_text, container, false)
15          textview = view.findViewById(R.id.textView1)
16          return view
17      }
18      fun changeTextProperties(fontSize : Int, text : String){
19          textview.setTextSize(fontSize.toFloat())
20          textview.setText(text)
21      }
22  }
```

Explanation:

Line 8 Line 13 to 16	For any fragment, it needs to extend the Fragment class. An inflater is needed to “inflate” the fragment’s layout into the given fragment container and this will be contained in a view variable and must be subsequently returned. Also note that in the XML file, the correct class is linked to it (next page)
Line 18 to 21	This method is called from MainActivity.kt when the button is clicked on.

fragment_text.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".TextFragment">

    <TextView
        android:id="@+id/textView1"
        android:layout_width="376dp"
        android:layout_height="228dp"
        android:text="@string/EditTextPrompt"
        android:textAlignment="center"
        android:textColor="@color/black"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.011" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

FragmentOne.kt

```
1   import android.content.Context
2   import android.os.Bundle
3   import androidx.fragment.app.Fragment
4   import android.view.LayoutInflater
5   import android.view.View
6   import android.view.ViewGroup
7   import android.widget.Button
8   import android.widget.EditText
9   import android.widget.SeekBar
10  import android.widget.Toast
11  import com.google.android.material.snackbar.Snackbar
12  import java.lang.ClassCastException
13
14  class FragmentOne : Fragment(), SeekBar.OnSeekBarChangeListener{
15      private var seekValue = 10
16      private lateinit var activityCallback : FragListener
17
18      override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
19          savedInstanceState: Bundle?): View? {
20          // Inflate the layout for this fragment
21          val view : View = inflater.inflate(R.layout.fragment_one, container, false)
22          val editText : EditText= view.findViewById(R.id.editText1)
23          val seekbar: SeekBar = view.findViewById(R.id.seekBar1)
24          val button : Button = view.findViewById(R.id.button1)
25
26          seekbar.setOnSeekBarChangeListener(this)
27          button.setOnClickListener{ view1 ->
28              activityCallback?.onButtonClick(seekValue, editText.text.toString()) }
29          return view
30      }
31      override fun onProgressChanged(p0: SeekBar?, p1: Int, p2: Boolean) {
32          seekValue = p1
33      }
34      override fun onStartTrackingTouch(p0: SeekBar?) {
35          Toast.makeText(activity, R.string.seekStart, Toast.LENGTH_SHORT).show()
36      }
37      override fun onStopTrackingTouch(p0: SeekBar?) {
```

```
38          Toast.makeText(activity, R.string.seekStop, Toast.LENGTH_SHORT).show()
39      }
40
41      public interface FragListener{
42          fun onButtonClick(position : Int, text :String)
43      }
44
45      override fun onAttach(context: Context) {
46          super.onAttach(context)
47          try{
48              activityCallback = context as FragListener
49          } catch (e : ClassCastException){
50              val str = context.toString()
51              throw ClassCastException("$str must implement FragListener")
52          }
53      }
54  }
```

Explanation:

Line 14 to 15 Line 26, 31 to 39	The code here describes how to implement a SeekBar. Note that as the fragment itself implements the SeekBar's listener, the methods, onProgressChanged(), onStartTrackingTouch() and onStopTrackingTouch() are directly overridden in the fragment class itself.
Line 14, 27 to 28 Line 41 to 53	The activityCallback is an instance of the app's (context) FragListener interface defined in the class itself. This listener will give FragmentOne a means to communicate the text properties defined in this fragment to the MainActivity which will be responsible for passing the information into TextFragment. Note that onAttach is called the moment the fragment is initialized and placed into its host activity in the fragment life cycle.

fragment_one.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".FragmentOne">

    <EditText
        android:id="@+id/editText1"
        android:layout_width="320dp"
        android:layout_height="51dp"
        android:text="@string/EditTextPrompt"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.494"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.08" />

    <SeekBar
        android:id="@+id/seekBar1"
        android:layout_width="322dp"
        android:layout_height="36dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
```

```
        app:layout_constraintHorizontal_bias="0.468"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/editText1"
        app:layout_constraintVertical_bias="0.046" />

    <Button
        android:id="@+id/button1"
        android:layout_width="162dp"
        android:layout_height="61dp"
        android:layout_marginTop="24dp"
        android:text="@string/BtnText"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.497"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/seekBar1"
        app:layout_constraintVertical_bias="0.0" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.kt

```
1   import android.content.Context
2   import android.os.Bundle
3   import android.view.MotionEvent
4   import android.view.inputmethod.InputMethodManager
5   import androidx.fragment.app.FragmentActivity
6   import com.google.android.material.snackbar.Snackbar
7
8   class MainActivity : FragmentActivity(), FragmentOne.FragListener {
9       override fun onCreate(savedInstanceState: Bundle?) {
10          super.onCreate(savedInstanceState)
11          setContentView(R.layout.activity_main)
12      }
13      // The method will force the touch keyboard to hide when user touches anywhere on screen
14      override fun onTouchEvent(event: MotionEvent?): Boolean {
15          val imm = getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
16          if(imm.isAcceptingText) imm.hideSoftInputFromWindow(currentFocus!!.windowToken, 0)
17          return true
18      }
19      override fun onButtonClick(position: Int, text: String) {
20          val textFragment = (supportFragmentManager.findFragmentById(R.id.fragment_text))
21                  as TextFragment
22          textFragment.changeTextProperties(position, text)
23          Snackbar.make(findViewById(R.id.overallLayout), R.string.BtnClick,
24                                      Snackbar.LENGTH_SHORT).setAction("Action",null).show()
25      }
26  }
```

Explanation:

Line 8 Line 19 to 25	The MainActivity will extend the FragmentActivity class, which will be required as FragmentOne and FragmentText are known otherwise as support-based fragments, and will require a supportFragmentManager to retrieve information about specific fragments (for this case, the TextFragment), so that its changeTextProperties function can be called. Note that FragmentOne's FragListener is implemented here so that the onButtonClick defined in FragmentOne can be overridden to retrieve the information from FragmentOne regarding its text properties to properly allow TextFragment to make the necessary changes and display the output.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/overallLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/mainText"
        android:textColor="@color/black"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.091" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_one"
        android:name="com.example.fragments.FragmentOne"
        android:layout_width="357dp"
        android:layout_height="174dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView"
        app:layout_constraintVertical_bias="0.048" />

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_text"
        android:name="com.example.fragments.TextFragment"
        android:layout_width="360dp"
        android:layout_height="328dp"
        android:layout_marginTop="24dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.489"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/fragment_one"
        app:layout_constraintVertical_bias="0.078" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

4.4 Floating Action Button (FAB) and SnackBar

Now that we know how to use fragments in our apps, we can make full use of fragments to create our apps. This section introduces the Floating Action Button (FAB) and the SnackBar.

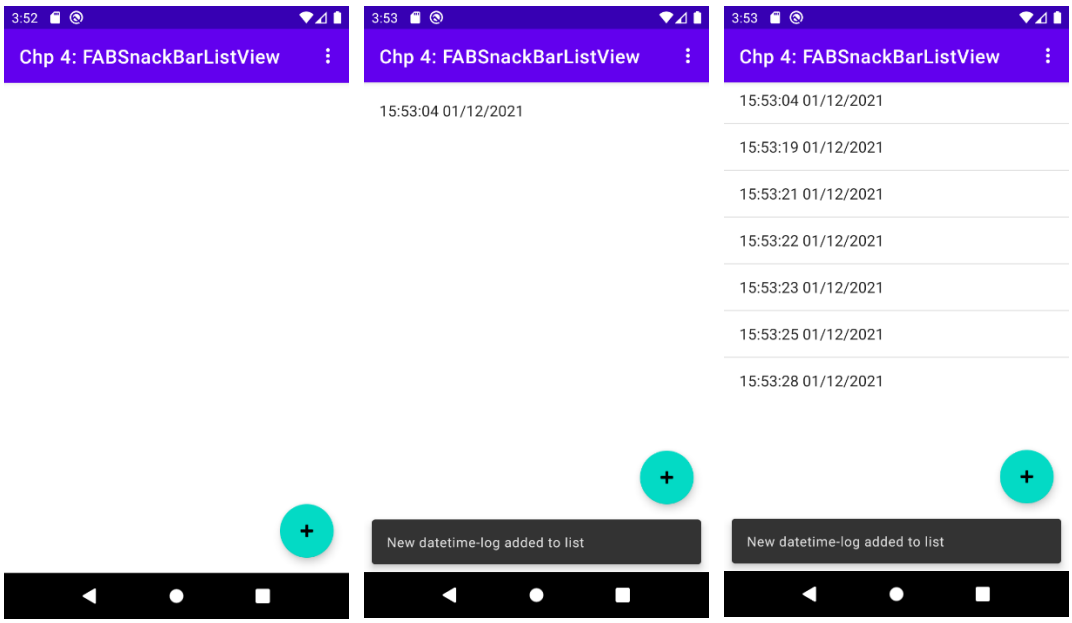
The FAB is a button which appears to float above the surface of the UI of an app and is generally used to promote the **most common action** within a UI screen. **Material design** guidelines state that an FAB **must be circular** and can be either 56x56 dp (default) or 40x40dp (mini) in size. It also should be positioned a **minimum of 16dp from the edge of the screen** on phones and 24dp on tablet devices. Regardless of the size, the button must contain an interior icon that is 24dpx24dp in size and **each UI screen should only have one FAB**. Common uses of the FAB include allowing user to add entries or send emails or to "morph" (with animations) to another element like a separate fragment containing other functionalities.

The SnackBar provides a way to present the user with information in the form of a panel that appears at the bottom of the screen. They contain a brief text message and an optional action button which will perform a task when tapped by the user. Once displayed, it will either timeout automatically or can be removed manually by the user via a swiping action. Similar to a Toast, the app will continue to function and respond to user interactions in the normal manner even while the SnackBar is present. The above descriptions will be made clearer in the following example:

Example 3: FABSnackBarListview

In the app, the FAB shows a + icon, indicating to the user that the FAB performs addition of entries. The moment the FAB is clicked, a timestamp will be added to a list with the SnackBar produced showing a short useful message regarding what just happened. Clicking on the FAB more times will result in more timestamps appended into the list.
Note that in this app, a fragment is used to contain the list of timestamps.

Screenshots of FABSnackBarListView app



strings.xml

```
<resources>
    <string name="app_name">Chp 4: FABSnackBarListView</string>
    <string name="action_settings">Settings</string>
    <string name="snackbarNote">New datetime-log added to list</string>
</resources>
```

menu_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.example.fabsnackbarlistview.MainActivity">
    <item
        android:id="@+id/action_settings"
        android:orderInCategory="100"
        android:title="@string/action_settings"
        app:showAsAction="never" />
</menu>
```

ListFragment.kt

```
1   import android.os.Bundle
2   import androidx.fragment.app.Fragment
3   import android.view.LayoutInflater
4   import android.view.View
5   import android.view.ViewGroup
6   import com.example.fabsnackbarlistview.databinding.FragmentListBinding
7
8   class ListFragment : Fragment() {
9
10      private var _binding: FragmentListBinding? = null
11      // This property is only valid between onCreateView and onDestroyView.
12      private val binding get() = _binding!!
13
14      override fun onCreateView(
15          inflater: LayoutInflater, container: ViewGroup?,  savedInstanceState: Bundle?): View? {
16          _binding = FragmentListBinding.inflate(inflater, container, false)
17          return binding.root
18      }
19      override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
20          super.onViewCreated(view, savedInstanceState)
21      }
22      override fun onDestroyView() {
23          super.onDestroyView()
24          _binding = null
25      }
26  }
```

Explanation:
Note that the ListFragment code above is the remnants after deleting what needs to be deleted from Android Studio's template Basic Activity. Also note the required databinding import, which is generated based on the associated XML file (for this case, fragment_list.xml). You may read more on the View Binding usage here: https://developer.android.com/topic/libraries/view-binding

For the list of timestamps, a ListView is used. (see below)

fragment_list.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ListFragment">

    <ListView
        android:id="@+id/listView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.037" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

MainActivity.kt

```
1   import android.os.Bundle
2   import com.google.android.material.snackbar.Snackbar
3   import androidx.appcompat.app.AppCompatActivity
4   import android.view.Menu
5   import android.view.MenuItem
6   import android.widget.ArrayAdapter
7   import android.widget.ListView
8   import com.google.android.material.floatingactionbutton.FloatingActionButton
9   import java.text.SimpleDateFormat
10  import java.util.*
11  import kotlin.collections.ArrayList
12
13  class MainActivity : AppCompatActivity() {
14      private var listItems = ArrayList<String>()
15      private lateinit var adapter: ArrayAdapter<String>
16      private lateinit var myListView : ListView
17
18      override fun onStart(){
19          super.onStart()
20          myListView = findViewById(R.id.listView)
21          adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, listItems)
22          myListView.adapter = adapter
23      }
24      override fun onCreate(savedInstanceState: Bundle?) {
25          super.onCreate(savedInstanceState)
26          setContentView(R.layout.activity_main)
27          setSupportActionBar(findViewById(R.id.toolbar))
28
29          val fab : FloatingActionButton = findViewById(R.id.fab)
30          fab.setOnClickListener { view ->
31              addListItem()
32              Snackbar.make(view, R.string.snackbarNote, Snackbar.LENGTH_LONG)
33                  .setAction("Action", null).show()
34          }
35      }
36      fun addListItem(){
37          val dateformat = SimpleDateFormat("HH:mm:ss dd/MM/yyyy", Locale.ENGLISH)
38          listItems.add(dateformat.format(Date()))
39          adapter.notifyDataSetChanged()
40      }
41      override fun onCreateOptionsMenu(menu: Menu): Boolean {
42          // Inflate the menu; this adds items to the action bar if it is present.
43          menuInflater.inflate(R.menu.menu_main, menu)
44          return true
```

```
45      }
46      override fun onOptionsItemSelected(item: MenuItem): Boolean {
47      // Handle action bar item clicks here. The action bar will automatically handle clicks on
48      // the Home/Up button, so long as you specify a parent activity in AndroidManifest.xml.
49          return when (item.itemId) {
50              R.id.action_settings -> true
51              else -> super.onOptionsItemSelected(item)
52          }
53      }
54  }
```

Explanation:

Line 14 to 16 Line 20 to 22 Line 36 to 39	An **Array Adapter** is used to contain the list of items which are stored in an ArrayList<String>. This array adapter will be the adapter for the ListView object which will display the timestamps. The addListItem() function will add a new timestamp into the ArrayList<String> and note that notifyDataSetChanged() must be called so that changes can be reflected in the ListView.
Line 29 to 34	The FAB will has an OnClickListener which will both add the new timestamp as well as display the SnackBar.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/Theme.FABSnackBarListView.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/Theme.FABSnackBarListView.PopupOverlay" />

    </com.google.android.material.appbar.AppBarLayout>

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/listFragment"
        android:name="com.example.fabsnackbarlistview.ListFragment"
        android:layout_width="match_parent"
        android:layout_height="532dp"
        android:layout_gravity="bottom" />

    <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_marginEnd="@dimen/fab_margin"
        android:layout_marginBottom="16dp"
        app:srcCompat="@android:drawable/ic_input_add" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

4.5 RecyclerView and CardView; ViewHolder

The RecyclerView and CardView widgets work together to provide scrollable lists of information to the user in which the information is presented in the form of individual cards. On an individual level, the CardView is simply only responsible for presenting information in cards which may otherwise be typically presented as a list. The RecyclerView will be more interesting to discuss.

Much like the ListView class, the purpose of the RecyclerView is to allow information to be presented to the user in the form of a scrollable list. The RecyclerView, however, provides a number of advantages over the ListView. In particular, the RecyclerView is significantly more efficient in the way that it **reuses existing views** that make up list items as they scroll off the screen **instead of creating new ones**. This increases the performance and reduces the resources used by a list, which is beneficial when presenting large amounts of data to the user.

In addition, the RecyclerView provides three built-in layout managers to control the way in which list items are presented to the user:

setLayoutManager(…)	**Description**
LinearLayoutManager(this)	The list items are presented as either a horizontal or vertical scrolling list.
GridLayoutManager(this)	The list items are presented in grid format. This manager is best used when the list items are of uniform size.
StaggeredGridLayoutManager(this)	The list items are presented in a staggered grid format. This manager is best used when the list items are not of uniform size.

Each list item displayed in a RecyclerView is created as an instance of the ***ViewHolder class***. The ViewHolder instance contains everything is necessary for the RecyclerView to display the list item, including the information to be displayed and the view layout used to display the item.

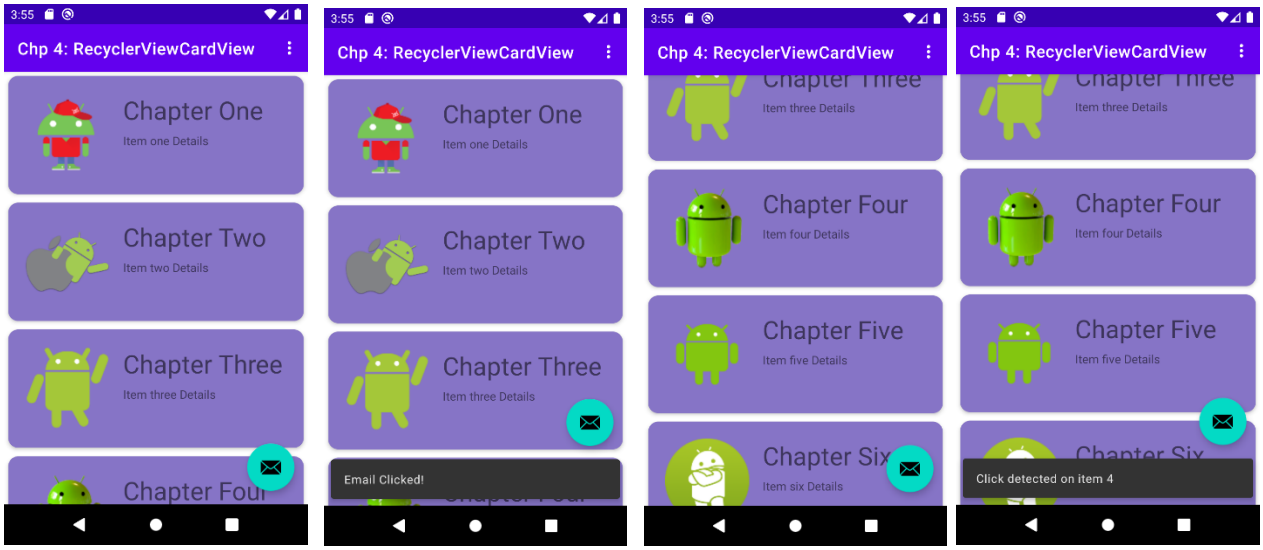
As with the ListView, the RecyclerView depends on an adapter to act as the intermediary between the RecyclerView instance and the data that is to be displayed to the user. The adapter is created as a subclass of the RecyclerView.Adapter class and must implement the following methods, which will be called at various points by the RecyclerView object to which the adapter is assigned:

RecyclerView Overriden Methods	**Description**
getItemCount()	Return a count of the number of the items that are to be displayed in the list.
onCreateViewHolder()	Creates and returns a ViewHolder object initialized with the view that is to be used to display the data. This view is typically created by inflating the XML layout file.
onBindViewHolder()	It is the responsibility of the onBindViewHolder() method to populate the views in the layout corresponding to the specified item, according to the model class defined, and to return the object to the RecyclerView where it will be presented to the user.

Example 4.1: RecyclerViewCardView

The app showcases a scrollable deck of cards, whereby each card displays an image, a chapter title and some details of the chapter (or rather item in this case).

Screenshots of RecyclerViewCardView app



strings.xml and menu_main.xml will not be shown as they contain only trivial code.

Chapters.kt

```
1   data class Chapters(val title: String, val detail: String, val images: Int)
```

Explanation:

Line 1	Kotlin allows for data classes to be created. This will be particularly useful to define model classes which you may need for your ViewModel. Note that this data class will already have all accessors and mutators implicitly defined.

card_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/card_view"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="5dp"
    app:cardBackgroundColor="#8674C6"
    app:cardCornerRadius="12dp"
    app:cardElevation="3dp"
    app:contentPadding="4dp"
    android:foreground="?selectableItemBackground"
    android:clickable="true">

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp">

        <ImageView
            android:id="@+id/item_image"
```

```
            android:layout_width="100dp"
            android:layout_height="100dp"
            app:layout_constraintLeft_toLeftOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/item_title"
            android:layout_width="236dp"
            android:layout_height="39dp"
            android:layout_marginStart="16dp"
            android:textSize="30sp"
            app:layout_constraintLeft_toRightOf="@+id/item_image"
            app:layout_constraintStart_toEndOf="@+id/item_image"
            app:layout_constraintTop_toTopOf="parent" />

        <TextView
            android:id="@+id/item_detail"
            android:layout_width="235dp"
            android:layout_height="52dp"
            android:layout_marginStart="16dp"
            android:layout_marginTop="8dp"
            android:textSize="14sp"
            app:layout_constraintLeft_toRightOf="@+id/item_image"
            app:layout_constraintStart_toEndOf="@+id/item_image"
            app:layout_constraintTop_toBottomOf="@+id/item_title" />

    </androidx.constraintlayout.widget.ConstraintLayout>
</androidx.cardview.widget.CardView>
```

Explanation:

The card_layout.xml essentially defines the layout for just one card. Note certain attributes you can edit to enhance the aesthetics of the card. The app:cardCornerRadius attribute affects how “rounded” you wish your vertices of your card to be. The app:cardElevation attribute affects how “high-up” from “ground-level” you want your card to be with respect to your app. Allowing your app to have depth improves the user experience by highlighting the relative importance of that component to the user in an intuitive way. If you want your cards to be clickable, ensure that android:clickable attribute is set to true.

RecyclerAdapter.kt

```
1   import android.view.View
2   import androidx.recyclerview.widget.RecyclerView
3   import android.widget.TextView
4   import android.view.ViewGroup
5   import android.widget.ImageView
6   import android.view.LayoutInflater
7   import com.google.android.material.snackbar.Snackbar
8
9   class RecyclerAdapter(val chpsList: ArrayList<Chapters>) :
10      RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {
11
12      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
13          val v : View = LayoutInflater.from(parent.context)
14              .inflate(R.layout.card_layout,parent,false)
15          return ViewHolder(v)
16      }
17      override fun onBindViewHolder(holder: RecyclerAdapter.ViewHolder, position: Int) {
18          holder.bindItems(chpsList[position])
19      }
20      override fun getItemCount() = chpsList.size
21
```

```
22      // The class holding the list view
23      class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
24          var itemImage: ImageView
25          var itemTitle: TextView
26          var itemDetails: TextView
27
28          init {
29              itemImage = itemView.findViewById(R.id.item_image)
30              itemTitle = itemView.findViewById(R.id.item_title)
31              itemDetails = itemView.findViewById(R.id.item_detail)
32
33              itemView.setOnClickListener{ view ->
34                  val pos = adapterPosition +1
35                  Snackbar.make(view, "Click detected on item $pos", Snackbar.LENGTH_LONG)
36                      .setAction("Action",null).show()
37              }
38          }
39          fun bindItems(chp : Chapters){
40              itemTitle.text = chp.title
41              itemDetails.text = chp.detail
42              itemImage.setImageResource(chp.images)
43          }
44      }
45  }
```

Explanation:

Line 9 to 20 Line 39 to 43	Note the compulsory methods which need to be overridden if you wish to extend the RecyclerView.Adapter class. The RecyclerAdapter is such that its constructor MUST have a parameter passed in, which will be the ArrayList containing the Chapter data, which will help in initialization of the RecyclerAdapter. Notice that the overridden methods have their specific purpose. The onCreateViewHolder() inflates the layout of each item to create a ViewHolder for each item. The onBindViewHolder() serves to bind each item of the ViewHolder to each item in the model. The getItemCount() simply just returns the total number of items in the adapter.
Line 23 to 38	A ViewHolder inner class is defined to hold the layout of each item such that when initialized, the attributes are linked to the respective views within the layout and a listener is implemented to define what to when the item is clicked.

content_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recycler_view"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
```

```
        app:layout_constraintVertical_bias="0.0" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Explanation:
The content_main.xml serves to hold the RecyclerView and will define the scrolling behaviour of the main content of the app itself. Hence, it is important to define app:layout_behaviour to ensure that the scrolling behaviour is implemented in your app.

MainActivity.kt

```
import android.os.Bundle
import com.google.android.material.snackbar.Snackbar
import androidx.appcompat.app.AppCompatActivity
import android.view.Menu
import android.view.MenuItem
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import com.google.android.material.floatingactionbutton.FloatingActionButton

class MainActivity : AppCompatActivity() {
    private val chpsList = ArrayList<Chapters>()

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

        val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)
        val layoutManager = LinearLayoutManager(this)
        recyclerView.layoutManager = layoutManager
        chpsList.add(Chapters("Chapter One", "Item one Details", R.drawable.android_image_1))
        chpsList.add(Chapters("Chapter Two", "Item two Details", R.drawable.android_image_2))
        chpsList.add(Chapters("Chapter Three", "Item three Details", R.drawable.android_image_3))
        chpsList.add(Chapters("Chapter Four", "Item four Details", R.drawable.android_image_4))
        chpsList.add(Chapters("Chapter Five", "Item five Details", R.drawable.android_image_5))
        chpsList.add(Chapters("Chapter Six", "Item six Details", R.drawable.android_image_6))
        chpsList.add(Chapters("Chapter Seven", "Item seven Details", R.drawable.android_image_7))
        chpsList.add(Chapters("Chapter Eight", "Item eight Details", R.drawable.android_image_8))
        val adapter = RecyclerAdapter(chpsList)
        recyclerView.adapter = adapter

        val fab = findViewById<FloatingActionButton>(R.id.fab)
        fab.setOnClickListener { view ->
            Snackbar.make(view, "Email Clicked!", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show()
        }
    }
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        // Inflate the menu; this adds items to the action bar if it is present.
        menuInflater.inflate(R.menu.menu_main, menu)
        return true
    }
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
    // 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.
        return when (item.itemId) {
            R.id.action_settings -> true
            else -> super.onOptionsItemSelected(item)
        }
    }
}
```

Explanation:

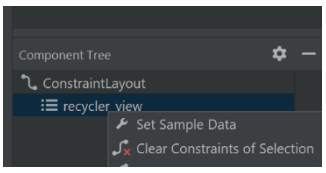
Line 19 to 30	In this app, the layout manager defined for the RecyclerView will be the LinearLayoutManager as seen in line 19 and 20. As the chpsList ArrayList is populated, this ArrayList is passed into the constructor of the RecyclerAdapter to link the model to the adapter itself.

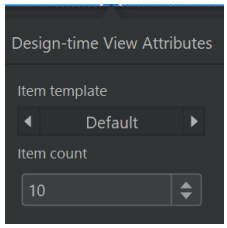
activity_main.xml

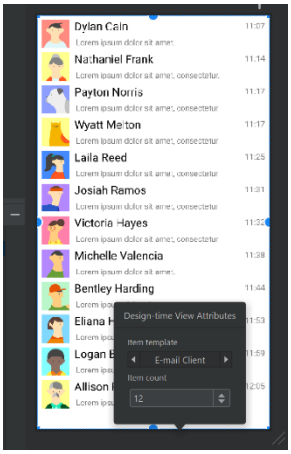
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/Theme.RecyclerViewCardView.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/Theme.RecyclerViewCardView.PopupOverlay" />

    </com.google.android.material.appbar.AppBarLayout>

    <include layout="@layout/content_main" />

    <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_marginEnd="@dimen/fab_margin"
        android:layout_marginBottom="16dp"
        app:srcCompat="@android:drawable/ic_dialog_email" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Explanation: You may want to take note of the highlighted code on how to include already made layouts into your current layouts. This is good for enforcing code reusability.

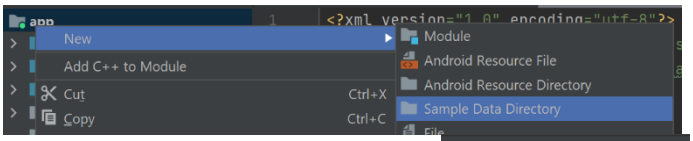
After creation of your app, you may wish to get an even better preview of how your app will look like with a sample of the populated data. You can go to content_main.xml and right click the recycler_view to "Set Sample Data" and a window will appear asking for your options and number of items. Changing the "Default" to other options will allow you to see how your app will look with some data.

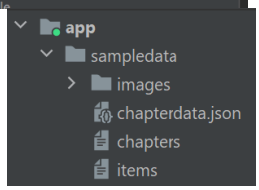




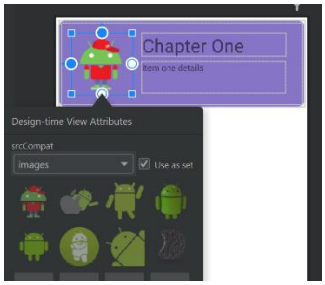


To switch to see what you intended, find the listItem attribute in Design mode and change the reference to refer to card_layout.xml. You will now see your recycler_view showing the cards, as intended, but without the data you want. You can create a reference the data by creating a new “Sample Data Directory” by right clicking on “app”. If it does not appear, you can manually create the sampledata folder and this is where you will store your custom sample data.

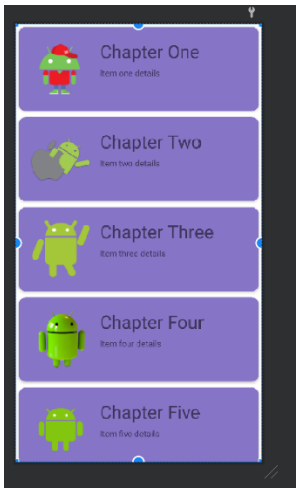




Load the card_layout.xml, go to each View and right-click to “Set Sample Data” and you will see your sample data (if you loaded them) for the images. For text, you have to reference the sample text data source. You will now see that your content_main.xml have all the loaded sample data in a nice card list.




```
<TextView
    android:id="@+id/item_title"
    android:layout_width="236dp"
    android:layout_height="39dp"
    android:layout_marginStart="16dp"
    android:textSize="30sp"
    app:layout_constraintLeft_toRightOf="@+id/item_image"
    app:layout_constraintStart_toEndOf="@+id/item_image"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="@sample/chapters" />
```



An alternative to the above is to use a JSON file to store all the data and subsequently edit card_layout.xml to refer to the respective keys using the tools:srcCompat attribute.

chapterdata.json

```
{
  "mydata": [
    {"chapter": "Chapter One",
    "details": "Item one details",
    "image": "@sample/images"},
    {"chapter": "Chapter Two",
      "details": "Item two details",
      "image": "@sample/images"},
    {"chapter": "Chapter Three",
      "details": "Item three details",
      "image": "@sample/images"},
    {"chapter": "Chapter Four",
      "details": "Item four details",
      "image": "@sample/images"},
    {"chapter": "Chapter Five",
      "details": "Item five details",
      "image": "@sample/images"},
    {"chapter": "Chapter Six",
      "details": "Item six details",
      "image": "@sample/images"},
    {"chapter": "Chapter Seven",
      "details": "Item seven details",
      "image": "@sample/images"},
    {"chapter": "Chapter Eight",
      "details": "Item eight details",
      "image": "@sample/images"}
  ]
}
```

```
<ImageView
    android:id="@+id/item_image"
    android:layout_width="100dp"
    android:layout_height="100dp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:srcCompat="@sample/chapterdata.json/mydata/image"/>

<TextView
    android:id="@+id/item_title"
    android:layout_width="236dp"
    android:layout_height="39dp"
    android:layout_marginStart="16dp"
    android:textSize="30sp"
    app:layout_constraintLeft_toRightOf="@+id/item_image"
    app:layout_constraintStart_toEndOf="@+id/item_image"
    app:layout_constraintTop_toTopOf="parent"
    tools:srcCompat="@sample/chapterdata.json/mydata/chapter"/>

<TextView
    android:id="@+id/item_detail"
    android:layout_width="235dp"
    android:layout_height="52dp"
    android:layout_marginStart="16dp"
    android:layout_marginTop="8dp"
    android:textSize="14sp"
    app:layout_constraintLeft_toRightOf="@+id/item_image"
    app:layout_constraintStart_toEndOf="@+id/item_image"
    app:layout_constraintTop_toBottomOf="@+id/item_title"
    tools:srcCompat="@sample/chapterdata.json/mydata/details"/>
```

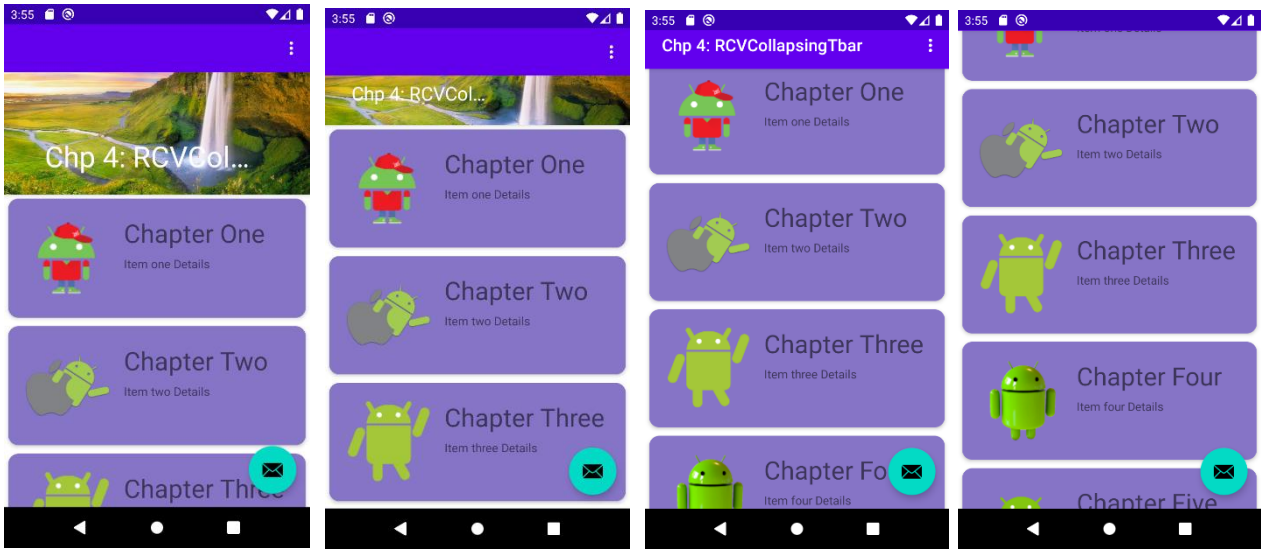
4.6 Collapsing Toolbar

This section will cover how an action bar can be customized to react to scrolling events on the screen. Specifically, we will be configuring the action bar to recede from view as an upward scrolling motion is imposed, only to reappear when a downward scrolling action is imposed.

Example 4.3: RecyclerViewCardViewCollapsingToolbar

This is a modification of Example 4.1 which includes a collapsing toolbar effect to the app as the user scrolls through the options in the RecyclerView.

Screenshots of RecyclerViewCardView app



To allow the collapsing toolbar to occur, only the activity_main.xml needs to be edited

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/Theme.RecyclerViewCardView.AppBarOverlay">

        <com.google.android.material.appbar.CollapsingToolbarLayout
            android:id="@+id/collapsing_toolbar"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:layout_scrollFlags="scroll|enterAlways"
            android:fitsSystemWindows="true"
            app:contentScrim="?attr/colorPrimary"
            app:expandedTitleMarginStart="48dp"
            app:expandedTitleMarginEnd="64dp">

        <ImageView
```

```
            android:id="@+id/backdrop"
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:scaleType="centerCrop"
            android:fitsSystemWindows="true"
            app:layout_collapseMode="parallax"
            android:src="@drawable/appbar_image"/>

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/Theme.RecyclerViewCardView.PopupOverlay"
            app:layout_scrollFlags="scroll|enterAlways"
            app:layout_collapseMode="pin"/>

        </com.google.android.material.appbar.CollapsingToolbarLayout>

    </com.google.android.material.appbar.AppBarLayout>

    <include layout="@layout/content_main" />

    <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_marginEnd="@dimen/fab_margin"
        android:layout_marginBottom="16dp"
        app:srcCompat="@android:drawable/ic_dialog_email" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Explanation:
We will need to add the CollapsingToolbarLayout which is a child of the AppBarLayout. We note that the CollapsingToolbarLayout has an attribute app:layout_scrollFlags which determine the scroll behaviour of the “collapse”. For this case, it is set to scroll OR enterAlways. The attribute has the following options.

app:layout_scrollFlags	Description
scroll	Indicates that the view is to be scrolled off the screen. If this is not set, the view will remain pinned at the top of the screen during scrolling events
enterAlways	When used in conjunction with *scroll*, an upwards scrolling motion will cause the view to retract. Any downward scrolling motion in this mode will cause the view to reappear
enterAlwaysCollapsed	When set on a view, that view will not expand from the collapsed state until the downward scrolling motion reaches the limit of the list. If the minHeight property is set, the view will appear during the initial scrolling motion but only until the minimum height is reached. It will then remain at that height and will not expand fully until the top of the list is reached. Note that this option only works when used in conjunction with BOTH *enterAlways* and *scroll*

exitUntilCollapsed	When set, the view will collapse during an upward scrolling motion until the minHeight threshold is met, at which point, it will remain at that height until the scroll direction changes.

Also note that in addition, an ImageView is placed as a backdrop for the scroll which will eventually disappear as a scroll is done deeper into the list. Note that the ImageView has an attribute app:layout_collapseMode. You can choose between none, pin and parallax.

Note also that the **toolbar** needs to have the same app:layout_scrollFlags set for the effect to be consistent, and that the toolbar needs to be a child of the CollapsingToolbarLayout in the activity_main.xml.

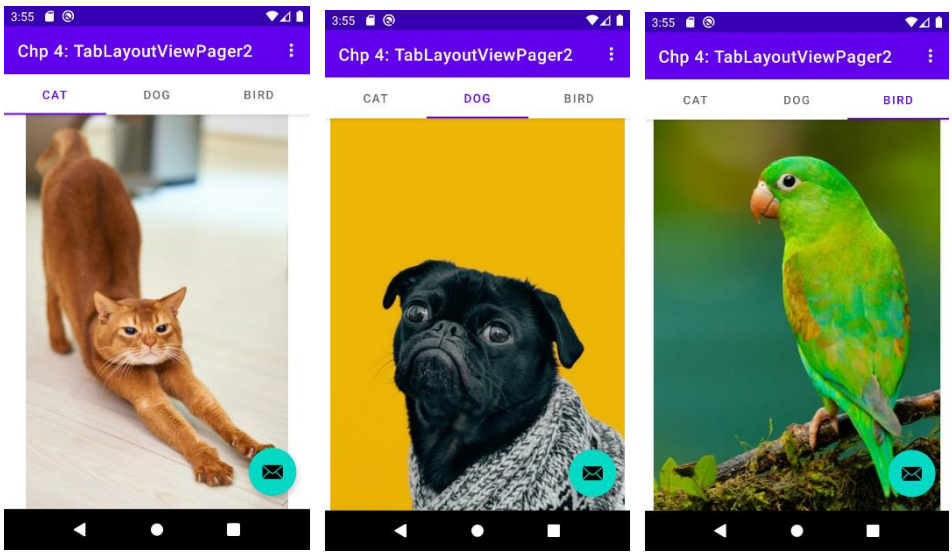
4.7 Using ViewPager2 in TabLayouts

The ViewPager2 is a useful companion class when used in conjunction with the TabLayout component to implement a tabbed UI in your app. The primary role of the ViewPager2 is to act as an adapter (similar to how RecyclerView works) to view your data in an organized format. You may also notice the "2" in ViewPager2, the reason being this is an upgrade from the original ViewPager, where ViewPager2 addresses many of the issues ViewPager originally had, most significantly supporting both vertical and horizontal paging, and right-to-left (RTL) paging.

Example 5: TabLayoutViewPager2

The app has three different tabs (Cat, Dog and Bird) and each tab will show its respective fragment to the user. The use will be able to either click on the tab to access it or do a swiping action to toggle between the tabs.

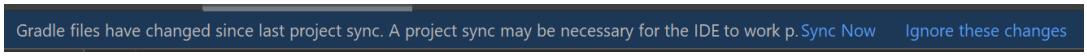
Screenshots of TabLayoutViewPager2 app



To be able to use the ViewPager2, you will need to first add the following implementation in the Gradle Scripts → build.gradle (Module: …), inside the **dependencies** block:

```
//ViewPager2 implementation
implementation 'androidx.viewpager2:viewpager2:1.0.0'
```

After adding the implementation code, ensure you select "Sync Now" on the top of the window.



Now you are ready to use ViewPager2. Note that each tab page is a fragment, and to make things a little less complicated, the fragments for the cat, dog and bird are similar, hence only one of the fragment codes and its associated XML file will be shown.

strings.xml and menu_main.xml will not be shown as they contain only trivial code.

BirdFragment.kt

```
1   import android.os.Bundle
2   import android.view.LayoutInflater
3   import android.view.View
4   import android.view.ViewGroup
5   import androidx.fragment.app.Fragment
6   import com.example.tablayoutviewpager2.databinding.FragmentBirdBinding
7
8   class BirdFragment : Fragment() {
9       private var _binding: FragmentBirdBinding? = null
10      private val binding get() = _binding!!
11
12      override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
13              savedInstanceState: Bundle?): View? {
14          _binding = FragmentBirdBinding.inflate(inflater, container, false)
15          val view = binding.root
16          return view
17      }
18  }
```

Explanation:

Line 9 to 17	This code uses the default code given in the Basic Activity layout, where its only purpose is the inflate the XML layout file into the FragmentContainerView

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".fragment.BirdFragment">

    <ImageView
        android:id="@+id/imageViewBird"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/bird"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.313" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

ViewPagerAdapter.kt

```
1   import androidx.fragment.app.Fragment
2   import androidx.fragment.app.FragmentManager
3   import androidx.lifecycle.Lifecycle
4   import androidx.viewpager2.adapter.FragmentStateAdapter
5   import com.example.tablayoutviewpager2.fragment.BirdFragment
6   import com.example.tablayoutviewpager2.fragment.CatFragment
```

```
7   import com.example.tablayoutviewpager2.fragment.DogFragment
8
9   private const val NUM_TABS = 3
10
11  class ViewPagerAdapter(fragmentManager: FragmentManager, lifecycle: Lifecycle) :
12          FragmentStateAdapter(fragmentManager, lifecycle) {
13
14      override fun getItemCount(): Int {
15          return NUM_TABS
16      }
17      override fun createFragment(position: Int): Fragment {
18          when (position) {
19              0 -> return CatFragment()
20              1 -> return DogFragment()
21          }
22          return BirdFragment()
23      }
24  }
```

Explanation:

Line 4 to 6 Line 14 to 23	This will be the code for the ViewPager2 adapter, which extends the FragmentStateAdapter which is found inside the ViewPager2 library (see imports). Note that you will have to also import the different fragments you wish to utilize for easy access programmatically. Note the overridden methods required (getItemCount(), createFragment()), which will allow you to link the corresponding “pages” of ViewPager2 to your created fragments (hence, the switch statement (when…) present)

MainActivity.kt

```
1   import android.os.Bundle
2   import com.google.android.material.snackbar.Snackbar
3   import androidx.appcompat.app.AppCompatActivity
4   import android.view.Menu
5   import android.view.MenuItem
6   import androidx.appcompat.widget.Toolbar
7   import androidx.viewpager2.widget.ViewPager2
8   import com.example.tablayoutviewpager2.adapter.ViewPagerAdapter
9   import com.google.android.material.floatingactionbutton.FloatingActionButton
10  import com.google.android.material.tabs.TabLayout
11  import com.google.android.material.tabs.TabLayoutMediator
12
13  class MainActivity : AppCompatActivity() {
14
15      private val animalsArray = arrayOf("Cat", "Dog", "Bird")
16
17      override fun onCreate(savedInstanceState: Bundle?) {
18          super.onCreate(savedInstanceState)
19          setContentView(R.layout.activity_main)
20          val toolbar : Toolbar = findViewById(R.id.toolbar)
21          setSupportActionBar(toolbar)
22
23          val viewPager = findViewById<ViewPager2>(R.id.viewPager)
24          val tabLayout = findViewById<TabLayout>(R.id.tab_layout)
25
26          val adapter = ViewPagerAdapter(supportFragmentManager, lifecycle)
27          viewPager.adapter = adapter
28
29          TabLayoutMediator(tabLayout, viewPager){ tab, position ->
30              tab.text = animalsArray[position]
31          }.attach()
```

```
32
33          val fab = findViewById<FloatingActionButton>(R.id.fab)
34          fab.setOnClickListener { view ->
35              Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
36                  .setAction("Action", null).show()
37          }
38      }
39      override fun onCreateOptionsMenu(menu: Menu?): Boolean {
40          // Inflate the menu; this adds items to the action bar if it is present.
41          menuInflater.inflate(R.menu.menu_main, menu)
42          return true
43      }
44      override fun onOptionsItemSelected(item: MenuItem): Boolean {
45      // Handle action bar item clicks here. The action bar will automatically handle clicks on
46      // the Home/Up button, so long as you specify a parent activity in AndroidManifest.xml.
47          return when (item.itemId) {
48              R.id.action_settings -> true
49              else -> super.onOptionsItemSelected(item)
50          }
51      }
52  }
```

Explanation:

Line 23 to 31	The setup of the ViewPager2 adapter in your activity is very similar to how the RecyclerView is set up in Example 4.1. You will have to ultimately like your TabLayout to your ViewPager2 adapter using the TabLayoutMediator, and you will have to link each of the titles of your tabs (the animalArray), to the corresponding pages within the ViewPager2 adapter that was setup in your ViewPagerAdapter class.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/Theme.TabLayoutViewPager2.AppBarOverlay">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/Theme.TabLayoutViewPager2.PopupOverlay"
            app:title="@string/app_name"
            app:titleTextColor="@color/white" />

        <com.google.android.material.tabs.TabLayout
            android:id="@+id/tab_layout"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:tabMode="fixed"
            app:tabGravity="fill"/>
```

```
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/viewPager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />

    <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_marginEnd="@dimen/fab_margin"
        android:layout_marginBottom="16dp"
        app:srcCompat="@android:drawable/ic_dialog_email" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Now you can implement tabbed features, scrollable features and fully utilize the action bar in your apps. How about implementing all of them at the same time in one app? Lab 4 will give you the opportunity to do so.

4.8 Using ViewPager2 in Onboarding

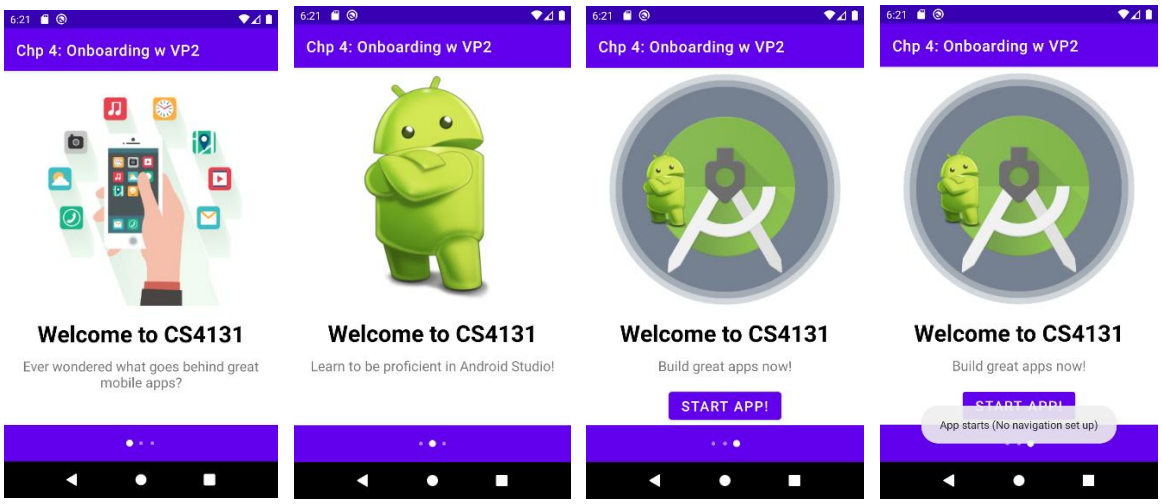
ViewPager2 can be used in scenarios which will require you to go from one page to another where each page is a fragment within in an activity. An application will be for onboarding.

Onboarding of an app means to show a **first-time user** how to get the most from your app, presenting some summarised information at app startup, which includes attention to noteworthy features of the app, or illustrating any required or recommended steps that users should take when using the app for the first time. The following example will highlight an implementation of onboarding using ViewPager2.

Example 6: Onboarding

The app basically is an onboarding for some CS4131 app. The user is able to swipe right or left to view different screens until the user reaches the right-most page and the user is allowed to click on a button (which should supposedly bring the user into the app but that is not implemented.

Screenshots of TabLayoutViewPager2 app



As ViewPager2 is used, recall in the previous section that you will need to include the ViewPager2 implementation in the build.gradle file, inside the **dependencies** block. We will also include the CircleIndicator, a custom view which provides for the onboarding circles at the bottom to indicate the which onboarding page is the user at.

AndroidManifest.xml additions into dependencies. Do not forget to “Sync Now”

```
// CircleIndicator
implementation 'me.relex:circleindicator:2.1.6'
// ViewPager2
implementation "androidx.viewpager2:viewpager2:1.0.0"
```

strings.xml

```
<resources>
    <string name="app_name">Chp 4: Onboarding w VP2</string>
    <string name="OBTitle">Welcome to CS4131</string>
    <string name="OBPg1">Ever wondered what goes behind great mobile apps?</string>
    <string name="OBPg2">Learn to be proficient in Android Studio!</string>
    <string name="OBPg3">Build great apps now!</string>
    <string name="start">Start App!</string>
</resources>
```

The fragments for the FirstFragment, SecondFragment and ThirdFragment are similar, hence only one of the fragment codes and its associated XML file will be shown.

FirstFragment.kt

```
1   import android.os.Bundle
2   import androidx.fragment.app.Fragment
3   import android.view.LayoutInflater
4   import android.view.View
5   import android.view.ViewGroup
6   import com.example.onboarding.R
7
8   class FirstFragment : Fragment() {
9       override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
10          savedInstanceState: Bundle?): View? {
11          return inflater.inflate(R.layout.fragment_first, container, false)
12      }
13  }
```

fragment_first.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="0dp"
        android:layout_height="300dp"
        android:layout_marginStart="20dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="20dp"
        android:src="@drawable/onboardingpg1"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
```

```
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/viewpaggertitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="@string/OBTitle"
        android:textColor="@color/black"
        android:textSize="30sp"
        android:textStyle="bold"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/imageView" />

    <TextView
        android:id="@+id/viewpaggerdescription"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="@string/OBPg1"
        android:textAlignment="center"

        android:textSize="18sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/viewpaggertitle" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

ViewPager2FragmentAdapter.kt

```
1   import androidx.fragment.app.Fragment
2   import androidx.fragment.app.FragmentActivity
3   import androidx.viewpager2.adapter.FragmentStateAdapter
4
5   class ViewPager2FragmentAdapter (FA: FragmentActivity,
6                   private val fragments:ArrayList<Fragment>): FragmentStateAdapter(FA) {
7
8       override fun getItemCount(): Int = fragments.size
9       override fun createFragment(position: Int): Fragment = fragments[position]
10  }
```

Explanation:

Note that the implementation is very similar to Example 5, except that this time, we enforce that a Fragment ArrayList is passed into the constructor of the adaptor for easy initialization.

MainActivity.kt

```
1   import androidx.appcompat.app.AppCompatActivity
2   import android.os.Bundle
3   import androidx.fragment.app.Fragment
4   import androidx.viewpager2.widget.ViewPager2
5   import com.example.onboarding.onboardingFragments.*
6   import me.relex.circleindicator.CircleIndicator3
7
8   class MainActivity : AppCompatActivity() {
9       private val fragmentList = ArrayList<Fragment>()
10      private lateinit var viewPager: ViewPager2
11      private lateinit var indicator: CircleIndicator3
12
13      override fun onCreate(savedInstanceState: Bundle?) {
14          super.onCreate(savedInstanceState)
15          setContentView(R.layout.activity_main)
16          castView()
```

```
17          fragmentList.add(FirstFragment())
18          fragmentList.add(SecondFragment())
19          fragmentList.add(ThirdFragment())
20
21          viewPager.adapter = ViewPager2FragmentAdapter(this, fragmentList)
22          viewPager.orientation = ViewPager2.ORIENTATION_HORIZONTAL
23
24          indicator.setViewPager(viewPager)
25      }
26
27      private fun castView() {
28          viewPager = findViewById(R.id.view_pager2)
29          indicator = findViewById(R.id.indicator)
30      }
31  }
```

Explanation:

Line 9 to 10 Line 16 to 22 Line 32	Notice that the setup for the fragments is quite different from how it is done in the TabLayout. First you will have to add the fragments into the defined Fragment ArrayList then simply set it as ViewPager2's adapter. Notice that you can also change the orientation as well.
Line 11 Line 24	The implemented CircleIndicator has a means to set up the links between the ViewPager2 and the indicators through their setViewPager() method, making it convenient.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <me.relex.circleindicator.CircleIndicator3
        android:id="@+id/indicator"
        android:layout_width="match_parent"
        android:layout_height="48dp"
        android:background="@color/purple_500"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

    <androidx.viewpager2.widget.ViewPager2
        android:id="@+id/view_pager2"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        app:layout_constraintBottom_toTopOf="@+id/indicator"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" >

    </androidx.viewpager2.widget.ViewPager2>

</androidx.constraintlayout.widget.ConstraintLayout>
```

The above only shows a very bare basic onboarding example. Onboarding pages can be vastly more complex than what you see in this example, and it will be up to you to find out more.

[Reference]

[1] Android Developers: Add the app bar: https://developer.android.com/training/appbar
[2] Android Menu Resource: https://developer.android.com/guide/topics/resources/menu-resource
[3] AppBarLayout https://developer.android.com/reference/com/google/android/material/appbar/AppBarLayout
[4] Android Developers: ArrayAdapter: https://developer.android.com/reference/android/widget/ArrayAdapter
[5] Android ViewPager2 with TabLayout: https://www.section.io/engineering-education/android-viewpager2/
[6] Migrating from ViewPager to ViewPager2: https://developer.android.com/training/animation/vp2-migration
[7] Simple way to implement an Onboarding slider using ViewPager2 https://medium.com/swlh/a-simple-way-to-implement-an-on-boarding-slider-using-viewpager2-in-android-b66b0ceea334

