Module 2

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 2 Study Guide and Deliverables

Topics:	• Basics of User Interface • Fragments
Readings:	• Online lectures • Notes and references cited in lectures • Chapters 5-11 and Appendix i from the "Head First" textbook (or corresponding chapters related to topics covered in this module in other textbooks)

Discussions:	• Discussion 2
	  ◦ Initial post due **Tuesday, July 19 at 6:00am ET**
	  ◦ Respond to threads posted by others due **Thursday, July 21 at 6:00am ET**

Labs:	• Lab 2 due **Tuesday, July 19 at 6:00am ET**
Assignments:	• Assignment 2 due **Tuesday, July 19 at 6:00am ET**
Assessments:	• Quiz 2 due **Tuesday, July 19 at 6:00am ET**
Live Classrooms:	• **Monday, July 18 from 6:00-8:00pm ET**

Learning Outcomes

By the end of this module, you will be able to:

- Compare views and viewgroups
- List basic widgets and layouts.
- Compare activities and fragments.
- Describe the life cycle of activities and fragments, and how each callback methods are called in their life cycles.
- Describe the basics of event handling in Android.
- Describe how to use intents for inter-component communication.

- Describe the Jetpack navigation component.
- Demonstrate basic skills of using an integrated development environment (Android Studio) and Android Software Development Kit (SDK) for implementing Android applications.
- Correctly implement a simple Android application with various layouts and widgets, as well as basic input event handling such as onClick.
- Correctly implement a simple Android application with several screen navigation using activities, fragments, intents and navigation.

Introduction

The user interface of an application consists of a collection of visual objects arranged on the screen that the user can interact with. UI design occurs in both requirement analysis and design phase. This usually involves:

- **Design each screen:** It is usually very helpful to create Lo-Fi(Low Fidelity) mockups using pen or paper, or digital wireframe tools such as balsamiq, Figma, Adobe XD, and etc. These Lo-Fi mockups are fast, cheap and help brainstorm and convey the ideas and get early feedback. In your project proposal, you should provide some Lo-Fi mockups for all your essential features.
- **Design screen navigation:** It is also important to show how UI screen changes based on user actions. A UI flow diagram or a state transition diagram can help visualize the UI navigation.

Compared with the UI design of desktop applications, mobile applications have unique **challenges**. For example, the mobile application UI should be easily adapted to different screen sizes and different orientations. As mobile devices in general are smaller than larger computers, the UI should be clean and easy to read. We also need to consider the performance. For example, we want to have screen transitions to be smooth without glitch. We may also need to consider the data leakage and security issues.

There are several tasks involved in the UI development:

1. Provide the look (what the app looks like). In the MVC architecture, this is usually provided by the View. In Android, different widgets and layout hierarchy defines the look.
2. Respond to the user's interactions. This is usually provided by the Control in the MVC architecture. In an Android application, the look is defined by various View and ViewGroup objects usually in a layout xml file, and then activities load xml files (the look) and define how to respond to the user interaction in the activity or fragment code.
3. Enable UI navigation: move from one screen to another.
4. Enable data communication between different screens.
5. We also need to consider how to maximize the code modularity, flexibility, maintainability and reusability. Fragments are usually used to achieve better reusability and flexibility.

> However, with the new *Android JetPack Compose toolkit*, we don't need to define a separate layout xml file. It also uses Kotlin code to define the look. It uses the idea of declarative UI. In this course, we will not discuss too much detail about Android JetPack Compose. If you want to use it, feel free to look at the Android developer website about it.

In this Module, you will learn how to develop an Android application with simple UI screens using Activity, View, ViewGroup, Fragment, Intent, and Navigation provided by the Android platform.

Activities

In the last module, we briefly introduced activities. An activity is a special component of an Android application. It is the entry point for the user interaction. Each activity is usually mapped to an application screen. To create an activity, we need to subclass the Activity class and override a very important method `onCreate()`. The `onCreate()` method creates the UI elements and draws them on the screen. This is usually done by specifying which layout to be used and inflating that layout on the screen. In the previous simple example, only a simple TextView is in the layout. In the following example, we will explore various widgets.

Test Yourself

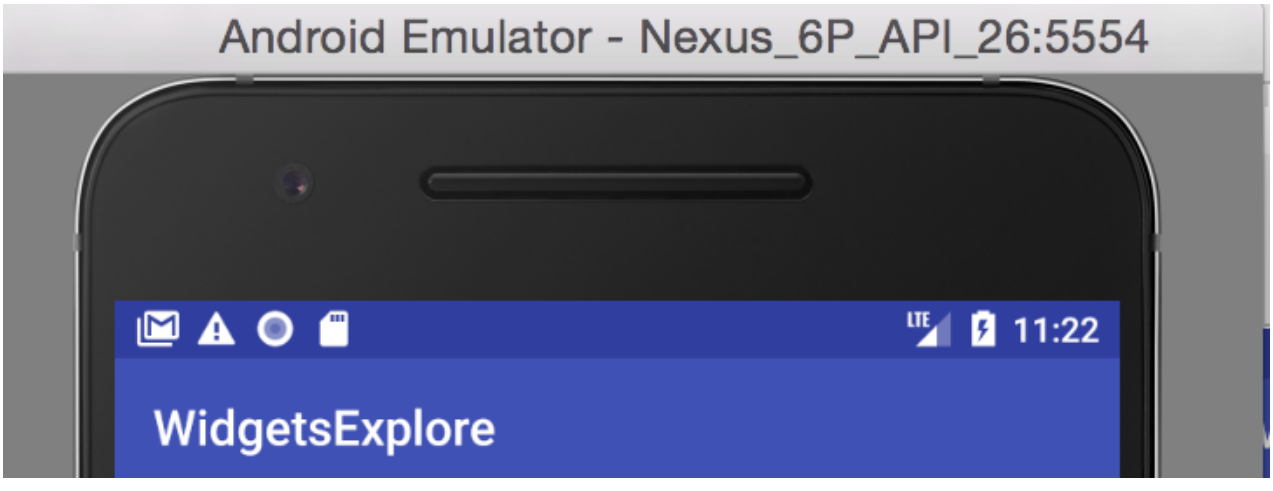
In which call back method of the Activity class should the layout be specified and inflated to the screen?

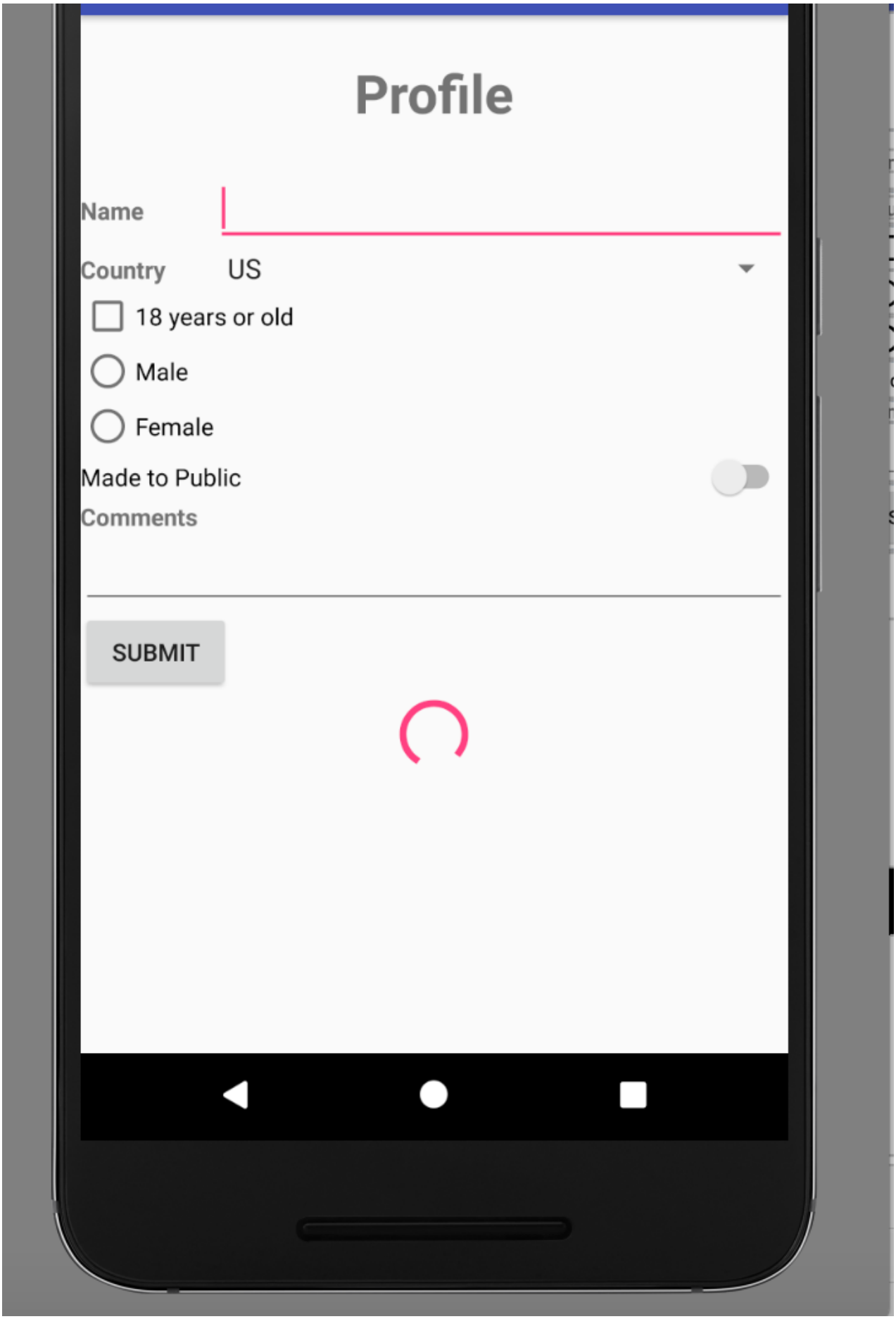
`onCreate()`

An Example - WidgetsExplore

Let us open the Android Studio and create a project "`WidgetsExplore`" to explore various View and ViewGroup objects. In this project, we will first use a simple layout `LinearLayout` instead of the default `ConstraintLayout`. WidgetsExplore is a simple application to get the user's profile. The application UI is shown as follows. The left figure shows the design. The right figure is the execution screen.


7:00
WidgetsExplore
Profile
Name
Country
US
18 years or old
Male
Female
Made to Public
Comments
SUBMIT


Android Emulator - Nexus_6P_API_26:5554
11:22
WidgetsExplore



In this project, we will use a number of widgets provided by Android: TextView, EditText, Button, Spinner, CheckBox, RadioButton, Switch, MultiLine EditText,and ProgressBar.

In the Project tool window, double click activity_main.xml and replace the existing code with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout 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"
  android:orientation="vertical"
  android:gravity="top"
  tools:context=edu.bu.widgetsexplore.MainActivity"
  android:weightSum="1">

   <TextView
       android:id="@+id/textViewId_Title"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:paddingTop="24dp"
       android:paddingBottom="24dp"
       android:textAlignment="center"
       android:textSize="14pt"
       android:text="Profile"
       android:textStyle="bold" />

   <LinearLayout
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:orientation="horizontal">

   <TextView
       android:id="@+id/textViewId_name"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_weight="0.4"
       android:text="Name"
       android:textStyle="bold" />

   <EditText
       android:id="@+id/editTextId_name"
       android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:layout_weight="3"
        android:inputType="textPersonName" />

    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

    <TextView
        android:id="@+id/textViewId_country"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0.4"
        android:textStyle="bold"
        android:text="Country" />

    <Spinner
        android:id="@+id/spinnerId_country"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="3"
        android:entries="@array/country_array" />
    </LinearLayout>

    <CheckBox
        android:id="@+id/checkBoxId_age"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="18 years or old" />

    <RadioGroup
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

    <RadioButton
        android:id="@+id/radioButtonId_male"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
    android:text="Male" />

<RadioButton
     android:id="@+id/radioButtonId_female"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Female" />

</RadioGroup>

<Switch
    android:id="@+id/switchId_public"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Made to Public" />

<TextView
    android:id="@+id/textViewId_comments"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textStyle="bold"
    android:text="Comments" />

<EditText
    android:id="@+id/editTextId_comments"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="textMultiLine" />

<Button
    android:id="@+id/buttonId_submit"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="submit" />

<ProgressBar
    android:id="@+id/progressBar"
    style="?android:attr/progressBarStyle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

```
</LinearLayou>
```

Test Yourself

Read the above XML file and find some common attributes of UI components. List at least three of them.

android:id, android:layout_width, android:layout_height

Test Yourself

Find errors in the following XML code:

```
<TextView
      Android:id = id/textViewId_comments
      Android:layout_width = wrap_content
      Android:layout_height = wrap_content
      Android:text = helloworld >
```

The name is case sensitive. Android -> android, need "" for all values, need id -> @+id. For example: android:id = " @+id/textViewId_comments"

Views

In this layout file, you may notice that all widgets and the linear layout used have some common properties such as id, layout_width, layout_height, etc. Actually, in Android, all GUI (graphical user interface) components are View objects. A View is an object that draws something on the screen that the user can interact with. It is a rectangular area of the display. Android provides this special class `View (android.view.View)` as the super class of all GUI widget classes.

Android provides a number of widgets. Each widget is defined by a `Widget` class in the `android.widget` package. Widgets inherit the basic attributes and methods from the `View` class and extend them. The View objects can set their attributes either in the layout file or by calling the corresponding methods in the Java code of the activity (or fragment) class.

The basic functionalities inherited from the `View` class are:

- Set/get basic properties such as id, layout_width, layout_ height, position, visibility, etc.
- Handle focus; respond to the moving focus
- Handle events; set event listeners to let View objects respond to events they listen to.

Here is a list of commonly used properties in the layout file: android:id, android:layout_width, android:layout_height, android:onclick, etc. The android:layout_width and android:layout_height attributes specify how wide and high the layout should

be. They must be set for all types of View objects. You can set their values to “match_parent,” “wrap_contant,” or a specific size such as 8dp.

> A dp (Density-independent Pixels) is an abstract unit that is based on the physical density of the screen - relative to a 160 dpi (dots per inch). When running on a higher density screen, the number of pixels used to draw 1dp is scaled up by a factor appropriate for the screen's dpi - likewise, when on a lower density screen, the number of pixels used for 1dp is scaled down.
>
> The ratio of dp-to-pixel will change with the screen density, but not necessarily in direct proportion. Using dp units (instead of px units) is a simple solution to making the view dimensions in your layout resize properly for different screen densities. In other words, it provides consistency for the real-world sizes of your UI elements across different devices.

(Source: Android Developer Website)

Here is a list of commonly used methods of various View classes: `findViewbyId()`, `getHeight()`, `getWidth()`, `setVisibility(int)`, `isFocused`, `requestFocus()`, `isClickable()`, etc.

For more details, please refer to Android's View webpage.

The following shows the class diagram of several widgets provided by Android:

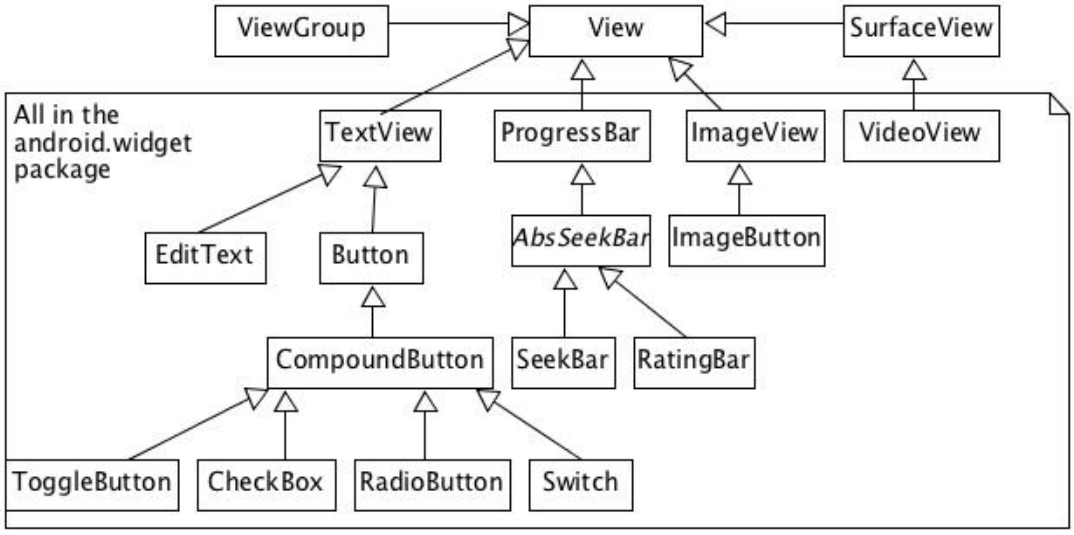
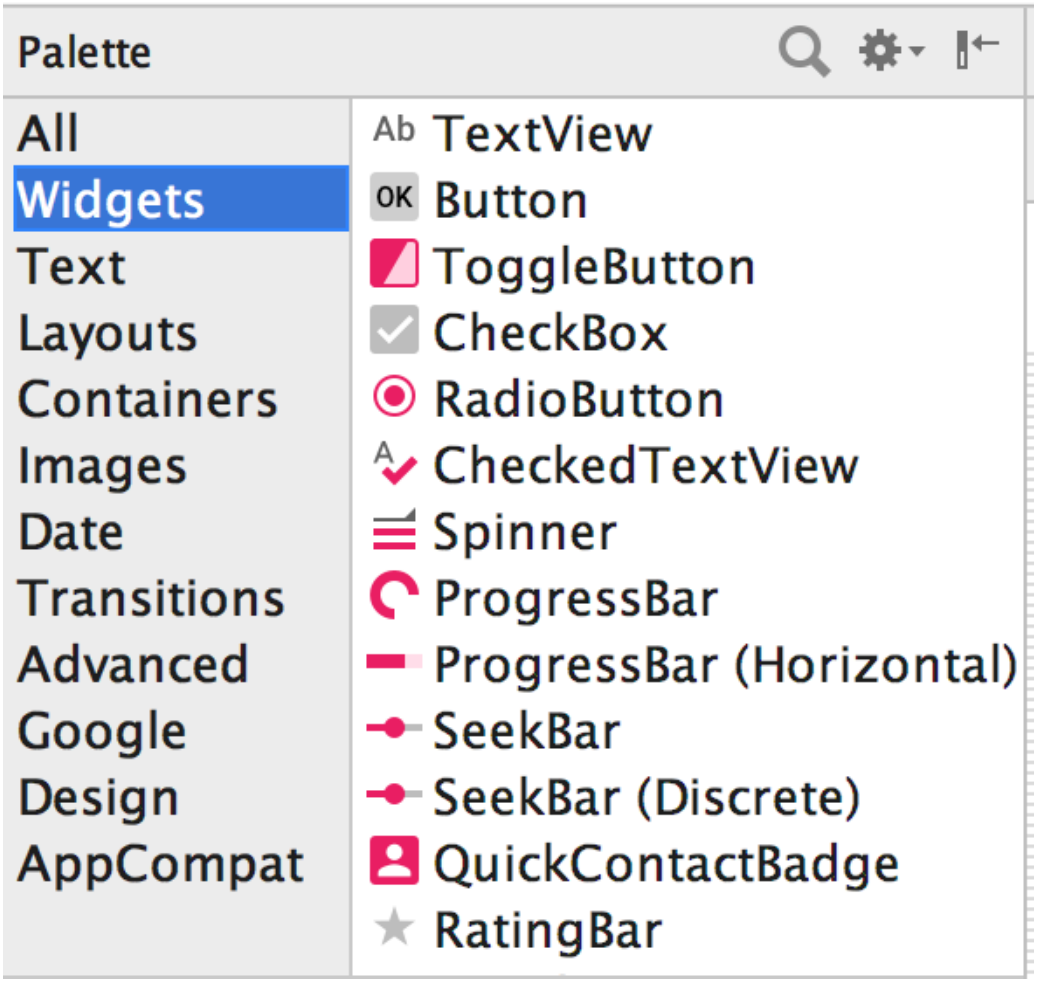


Figure 1.1 Class Diagram of Android Views

Some commonly used widgets are explained below:



Screenshot of Palette in AS

Text I/O

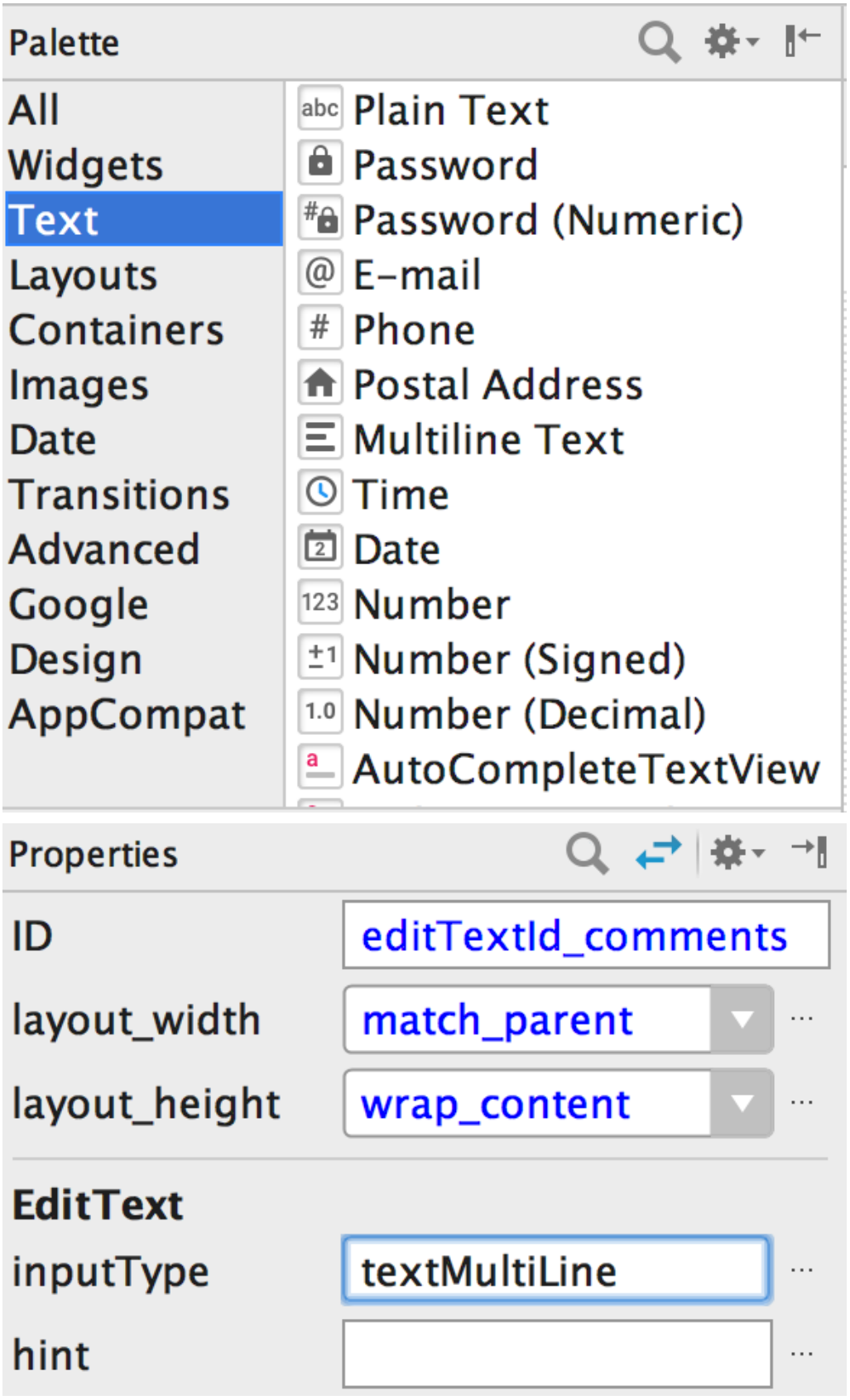
- TextView: mainly used to display text to the user. Properties include `text`, `textColor`, `textSize`, `textStyle`, `fontFamily`, etc.
  In the WidgetsExplore app, we used several TextViews to display labels, which are added inside the layout XML file. For example, a “Name” label is added as follows:

```
<TextView
    android:id="@+id/textViewId_name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="0.4"
    android:text="Name"
    android:textStyle="bold" />
```

- EditText: a subclass of TextView with rich editing capabilities. It inherits lots of attributes from TextView. When given focus, a cursor will automatically appear and a soft keyboard will be displayed. EditText can be configured for single or multi-line editing.
  - The `inputType` property can be set with different values: `textPersonName`(Plain Text), `textPassword`, `numberPassword`, `textEmailAddress`, `phone`, `textPostalAddress`, `textMultLine`, `time`, `date`, `number`,

etc. Further refinement includes `textCapSentences`, `textCapCharacters`, `textCapWords`, `textAutoCorrect`, `textAutoComplete`, `textNoSuggestions`, `TextURI`.

The following figure shows the `Multiline Text` is basically an `EditText` with the `inputType` property set to `textMultiLine`:



Screenshot of EditText Widget Properties

In the WidgetsExplore app, we used two types of textview. One is used to input the user name, another is used to input the comments. Each has a different inputType. Use the following code in the xml file to add a name field.

```
<EditText
      android:id="@+id/editTextId_name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="3"
      android:inputType="textPersonName" />
```

Use the following code in the XML file to add a multiline comment:

```
<EditText
       android:id="@+id/editTextId_comments"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:ems="10"
       android:inputType="textMultiLine" />
```

Buttons

- Button: it is also a subclass of TextView. It inherits the properties such as text and the methods such as setText(). In addition, it responds to a click event. You need to set a click listener on the Button object in the corresponding Activity Java code, or set the **onClick** attribute in the Layout XML file. We will discuss how to handle events in the later section. Find the button element defined in the previous xml file.
- CompoundButton: it is a subclass of Button. There are several different compound buttons that are subclasses of CompoundButton.
    - ToggleButton: displays checked/unchecked states as a button with a "light" indicator and by default accompanied with the text "ON" or "OFF".
    - Switch: a two-state toggle switch widget that can select between two options.
    - CheckBox: a two-state button that can be either checked or unchecked.
    - RadioButton: a two-state button that can be either checked or unchecked. RadioButtons are normally used together in a RadioGroup as shown in the WidgetsExplore project. Only one RadioButton can be checked in a RadioGroup.

Bars

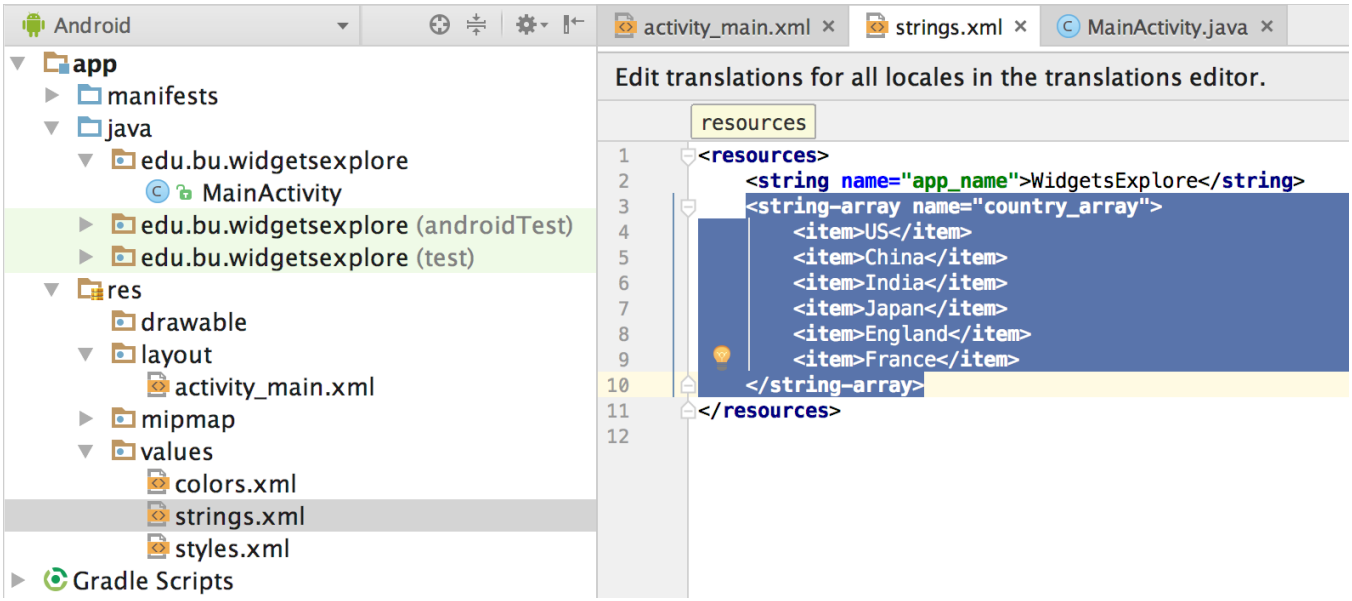
- ProgressBar: display a progress bar to a user in a non-interruptive way. Show the progress bar in your app's user interface or in a notification instead of within a dialog.
- SeekBar: an extension of ProgressBar that adds a draggable thumb. The user can touch the thumb and drag left or right to set the current progress level or use the arrow keys. Placing focusable widgets to the left or right of a SeekBar is discouraged.
- RatingBar: an extension of SeekBar and ProgressBar that shows a rating in stars. The user can touch/drag or use arrow keys to set the rating when using the default size RatingBar.

Dropdown Box

- Spinner: it is a dropbox that you can select one value from a set. In the default state, a spinner shows its currently selected value. Touching the spinner displays a dropdown menu with all other available values, from which the user can select a new one. It is actually not a subclass of View, instead it is a subclass of ViewGroup which we will discuss later. The set of value displayed can be set directly through the property `android:entries` as:

android:entries="@array/country_array"

Country_array is a string array defined in strings.xml as follows:



Screenshot of strings.xml

ImageView & VideoView

- ImageView: displays image resources, for example Bitmap or Drawable resources. ImageView is also commonly used to apply tints to an image and handle image scaling..
- VideoView: an extension of a SurfaceView, used to display a video file.

Defining Widgets in the XML File

Normally, the properties of a View object are set in the layout XML file **statically**. For example:

```
<TextView
     android:id="@+id/textViewId_name"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_weight="0.4"
```

```
    android:text="Name"
    android:textStyle="bold" />
```

Using Widgets in the Java/Kotlin code

We can manipulate (create, read, change, delete) each individual widget in Java/Kotlin code dynamically. To do this, we need to first get the reference of that widget object. This can be done usually based on its id by calling the method `findViewById()`, and then calling the correponding methods of that widget. The method `findViewById()` is defined in both Activity class and View class. It can be called in an Activity code, or called by a view object to find its children view objects. For example, the following code snippet shows how to get the name typed by the user in the `EditText` view object in the WidgetsExplore application. The `id` of that `EditText` is defined in the XML file as `editTextId_name`. Since the method `findViewById()` will return a `View` type, you need to specify it as an `EditText` type. Then get its typed text through its attribute text (in Kotlin, or call the method `getText()` in Java) to get the typed text and call the method `toString()` to convert the text into a string. Be aware that the types of the two variables used below are inferrable, so no need to declare their type explicitly.

```
val nameEditText = findViewById<EditText>(R.id.editTextId_name)
var name = nameEditText.text.toString();
```

Test Yourself

Look at the class diagram of Android View classes, and answer the following questions: what is the direct superclass of the Button class? What is the relationship between the View class and the ViewGroup class?

TextView is the superclass of the Button class. ViewGroup is a subclass of View.

Test Yourself

The _____ method is used to obtain the reference to a view widget object in Activity by its id.

findViewById()

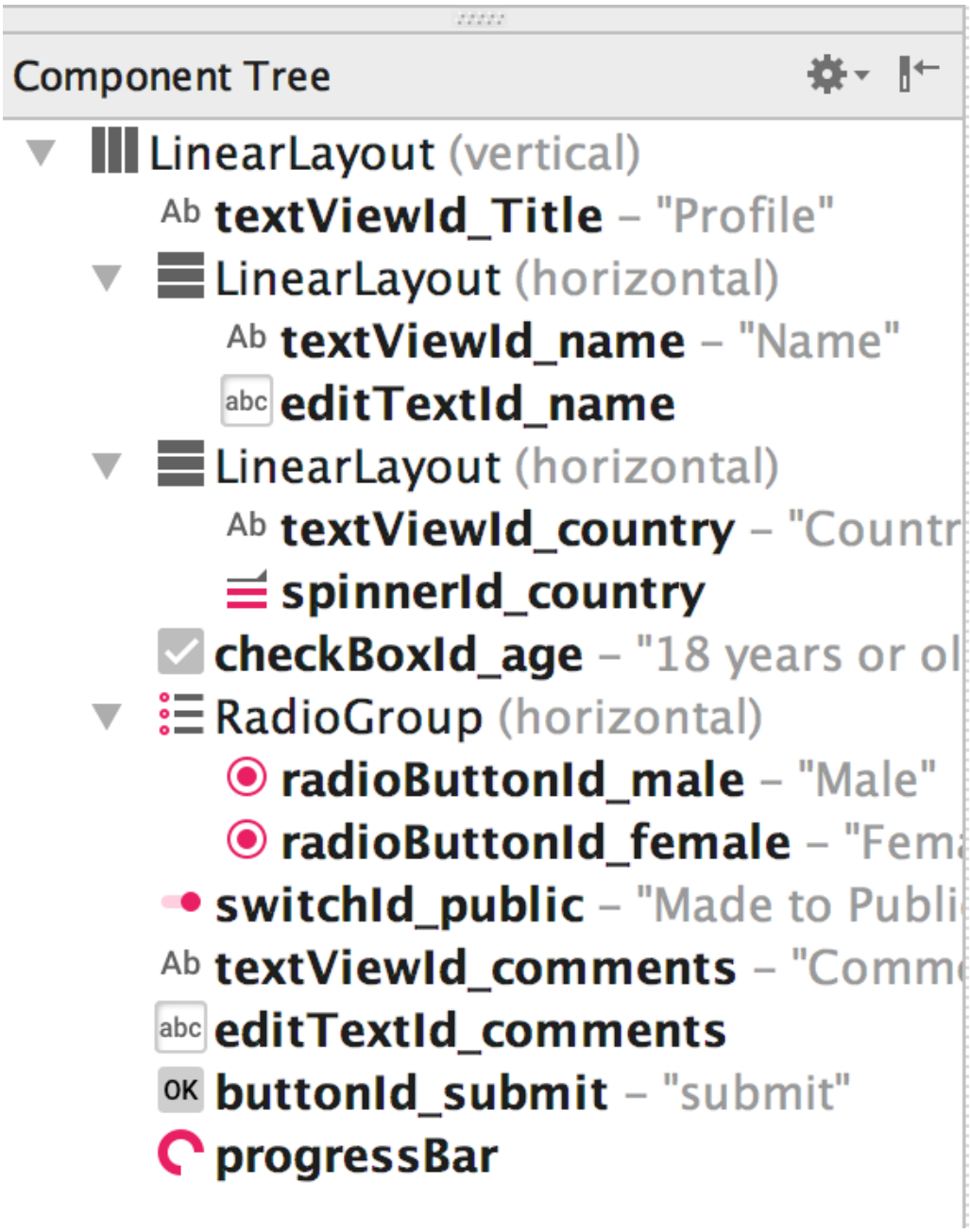
Test Yourself

Which widget can you use to display a list of data in a dropbox?

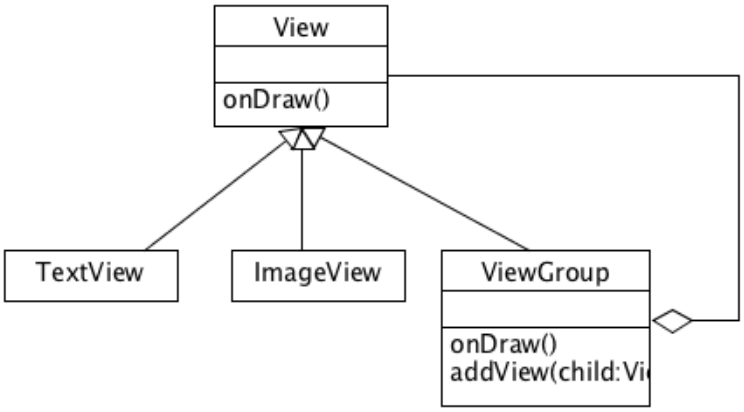
Spinner

ViewGroups & Layouts

The Layout xml file defines what the screen looks like. A Layout is a ViewGroup. A ViewGroup is an object that holds other View (and ViewGroup) objects in order to define the layout of the interface. As shown in the component tree of the WidgetsExplore project as below, the top level LinearLayout has multiple children in it, including simple widgets like TextView, EditText, as well as two other LinearLayouts, which further consists of other view objects.

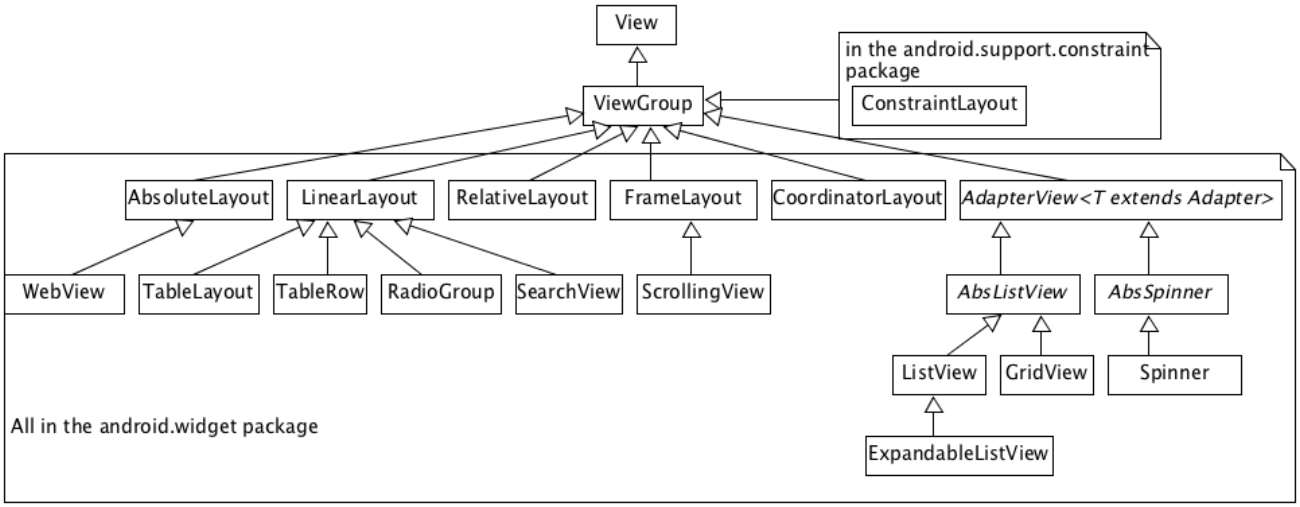


Interestingly, the ViewGroup class is also a subclass of View. Android uses the composite pattern in the View and ViewGroup design, so that both View and ViewGroup can be treated uniformly. The following class diagram shows this tree hierarchy relationship presented by the composite pattern.



Class Diagram of the Composite Pattern Used in Android View Elements

A layout defines the visual structure of a user interface. It is subclassed from ViewGroup. It defines View components in it, along with their attributes and relationships. Another type of ViewGroup is the container widgets such as RadioGroup and ListView. The following class diagram shows some subclasses of ViewGroup.



Class Diagram of ViewGroup classes

In the above WidgetsExplore project, we only used LinearLayout. There are several other types of layouts which are discussed below.

LinearLayout

A linear layout displays its children view objects next to each other in a single direction, either vertically or horizontally, which is set through the orientation property.

- The display order of the children view objects is the same order in which they are defined.
- Since a LinearLayout is an indirect subclass of View, it inherits properties like id, layout_width, layout_height.
- Use the `android:layout_weight` property of each child to specify how much of the layout space that child should occupy relative to other children.
- Use the `android:gravity` property to specify how to position the content of a View, e.g. the position of the text in a text field.

- Use the `android:layout_gravity` to specify where a view should appear in its parent linearlayout.

The linear layout is mainly used for simple arrangements.

For more details, refer to Linear Layout.

```
<LinearLayout
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:orientation="horizontal">

 <TextView
      android:id="@+id/textViewId_name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="0.4"
      android:text="Name"
      android:textStyle="bold" />

<EditText
      android:id="@+id/editTextId_name"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="3"
      android:inputType="textPersonName" />

</LinearLayout>
```

TableLayout

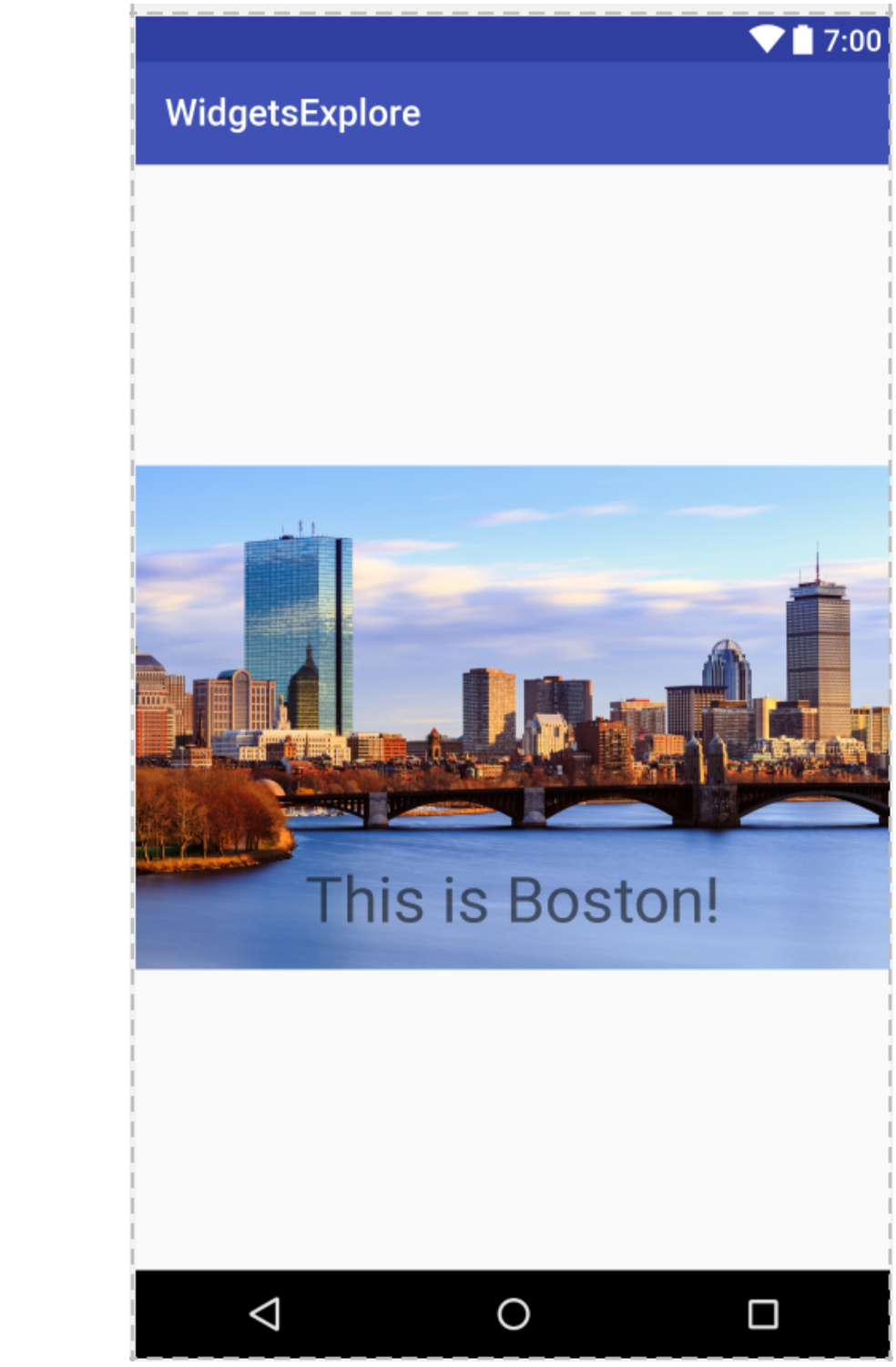
A table layout displays child View elements in rows and columns. Each row within a table is represented by a TableRow child object, which, in turn, contains a view object for each cell. It is similar to the HTML table. For more details, please refer to Android's Table webpage.

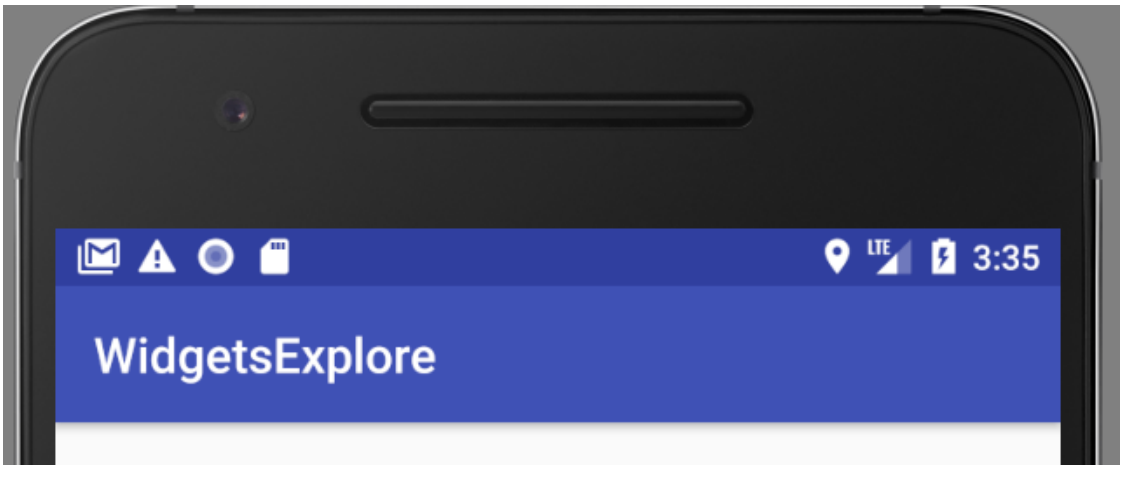
FrameLayout

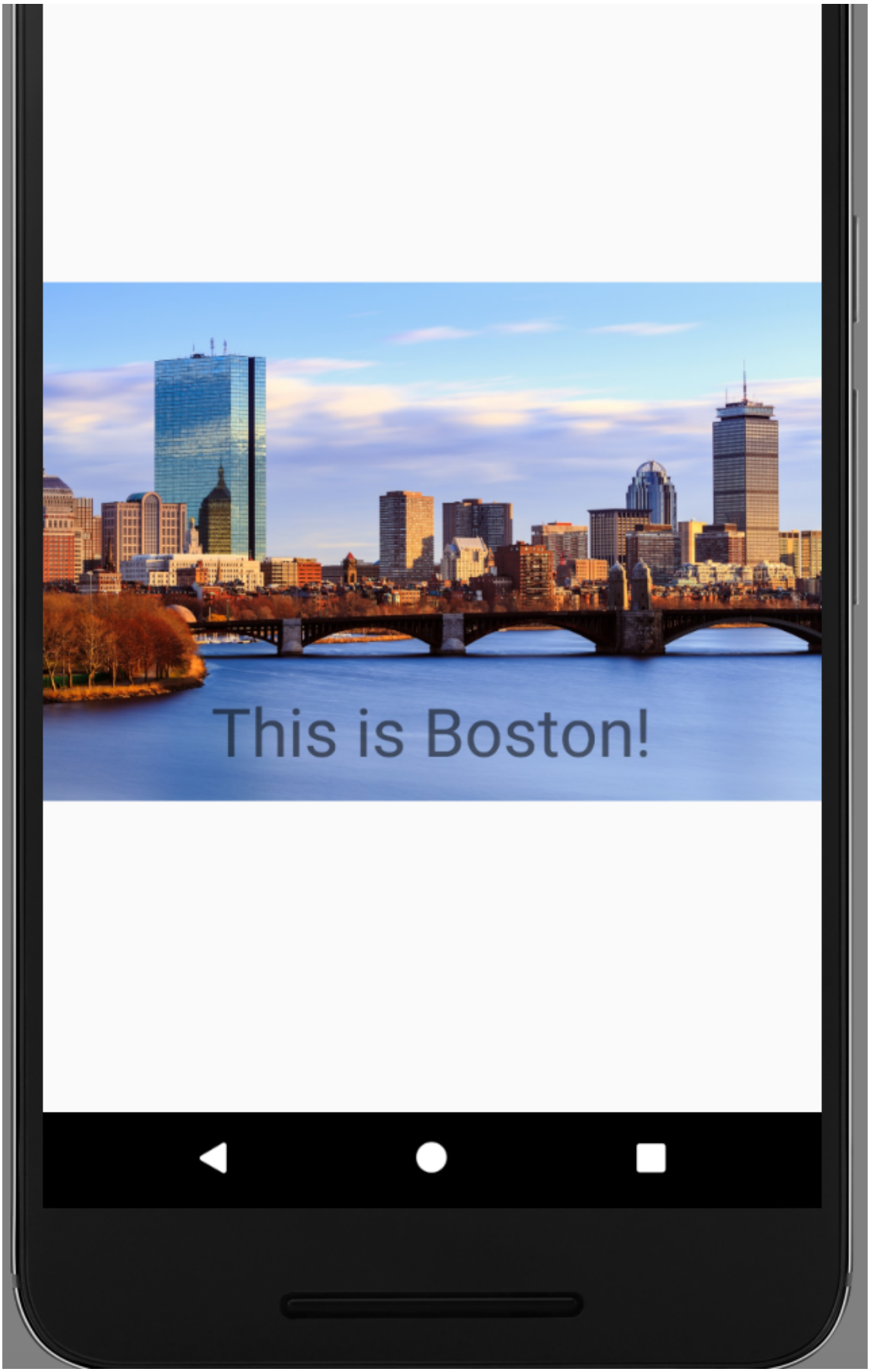
A frame layout stacks its child view elements on top of each other, in the order they are defined. This allows you to, for example, display text on top of an image. You can also use `layout_gravity` to specify where a view appears. It is also commonly used to display fragments. Here is an example:

```xml
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout 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="edu.bu.widgetsexplore.frameLayoutActivity">


    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:src="@drawable/boston" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textSize="16pt"
        android:paddingTop="200dp"
        android:text="This is Boston!"/>
</FrameLayout>
```

The following screenshots show the design view on the left and the execution result on the right.


7:00
WidgetsExplore
This is Boston!


3:35
WidgetsExplore



RelativeLayout & ConstraintLayout

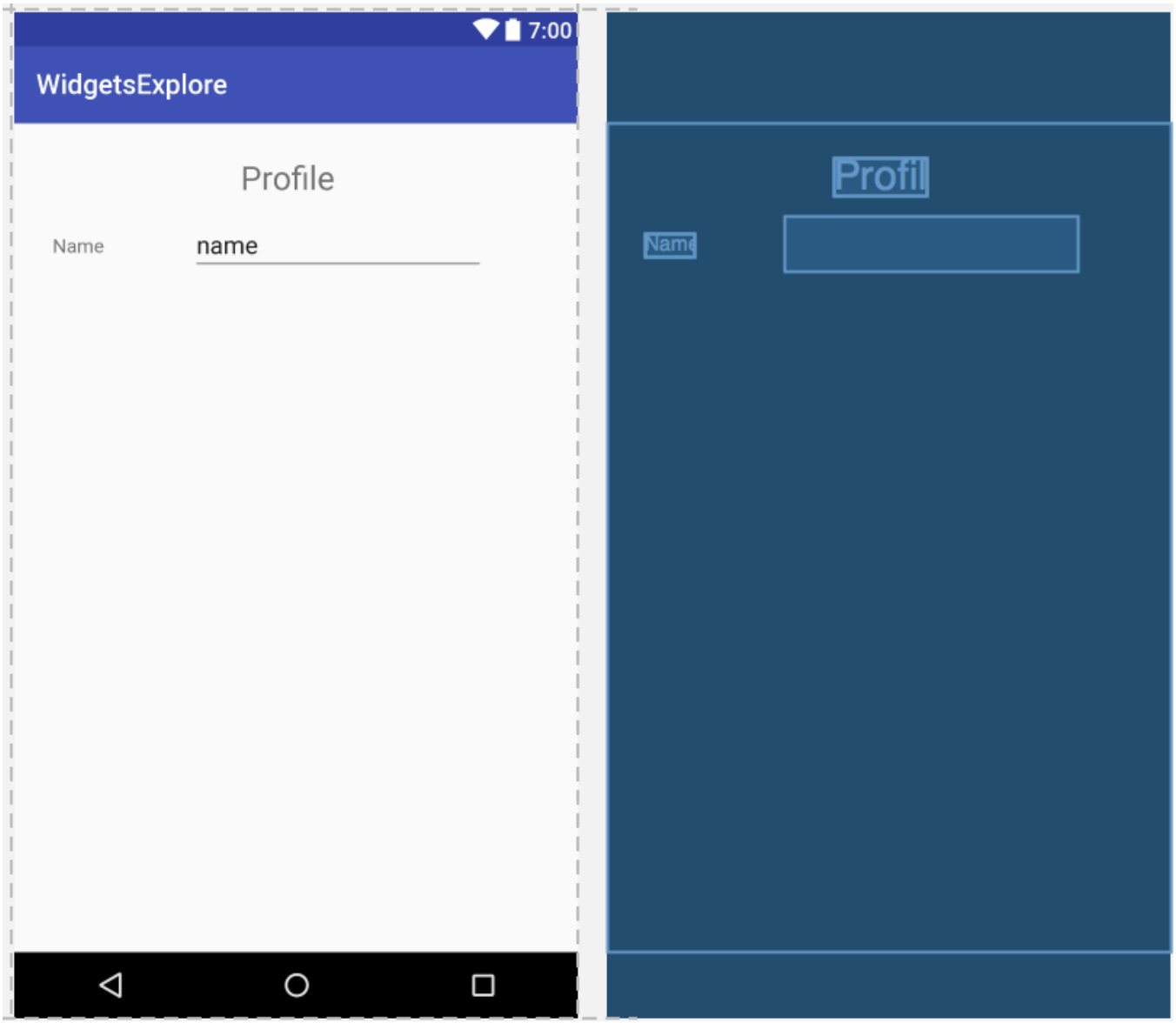
In this example, currently we use nested LinearLayouts to achieve a complex layout. However, Android provides two other types of layout that can achieve this flexibility without nesting: RelativeLayout and ConstraintLayout. Both can position the child view elements according to relationships between sibling views and the parent layout. They are more useful for a variety of screen sizes and orientations.

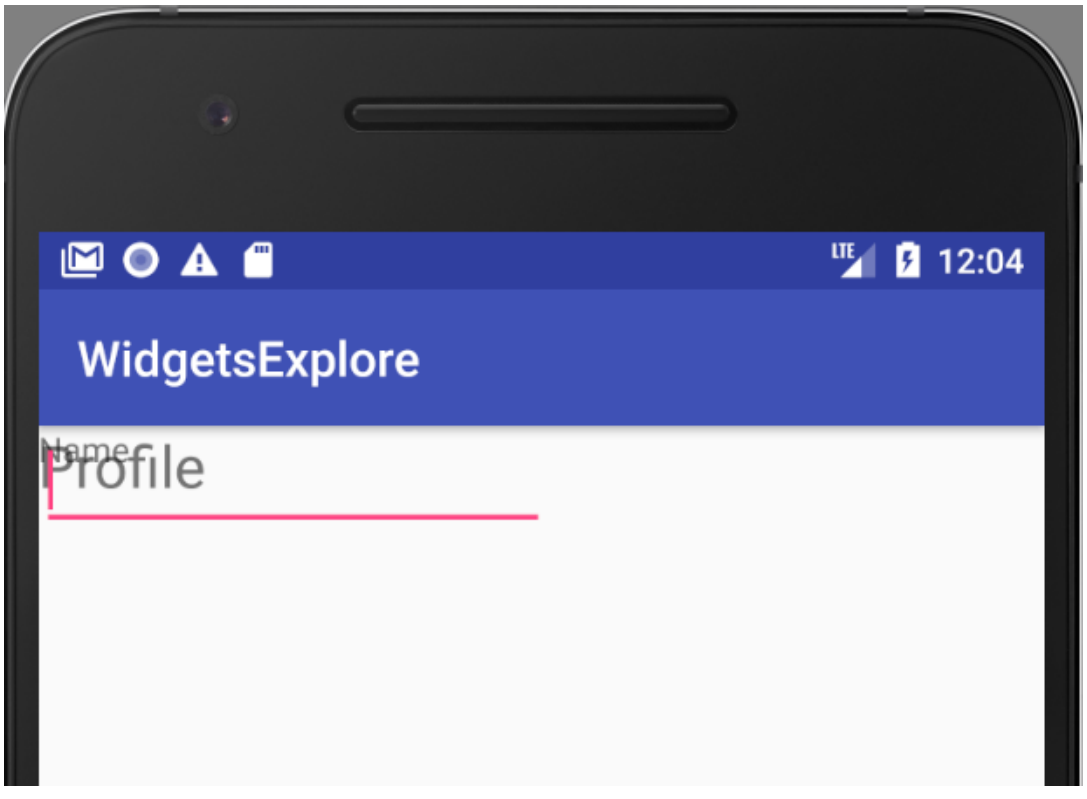
RelativeLayout is provided by the original Android Framework. When using RelativeLayout, the relative position layout attributes used by the containing views include `layout_below, layout_alignParentLeft, layout_toLeftOf,` etc.

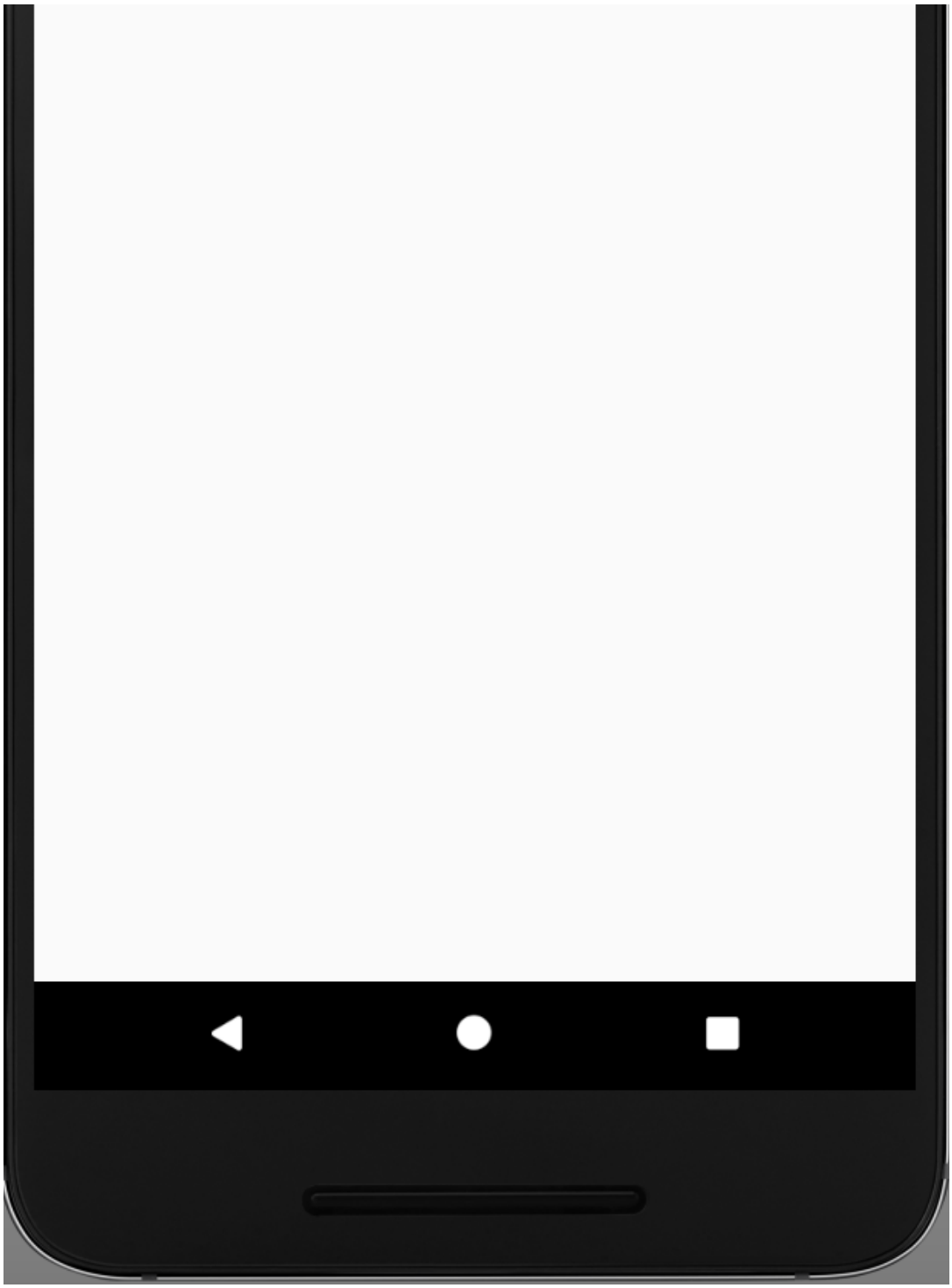
However, ConstraintLayout is available as a support library that you can use on Android systems starting with API level 9 (Gingerbread). In the new version, **AndroidX replaces the original support library APIs with packages in the Androidx namespace**. ConstraintLayout also replaces RelativeLayout in the templates provided by Android Studio 2.3 or above. You can build constraint layouts using visual tools in Android studio 2.3 or above. The visual drag and drop tool makes it easier to use the constraint layout.

Now, let us create a new Layout resource file and use ConstraintLayout for the same profile page we designed for the WidgetsExplore app.

When using ConstraintLayout, the position of a view is ***irrespective* of where the view is positioned in the blueprint area, it is also *irrespective* of where the view is positioned in the XML** file. For example, the following screenshot on the left side shows how the widgets are positioned in the design editor window, and the right side shows the execution screen. While they are correctly positioned in the design editor window, they overlap on the screen when you run the app, because all widgets are by default positioned at the top left corner of the screen if no constraints are specified.


7:00
WidgetsExplore
Profile
Name
name
Profil
Name


12:04
WidgetsExplore
Name
Profile



To define a view's position in the ConstraintLayout, you must add **at least one horizontal and one vertical constraint for the view object**. Each constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline.

Please read more details about how to use the constraint layout on the android developer website at: Build a Responsive UI with ConstraintLayout. Here is an example of the XML layout file using ConstraintLayout shown below.

```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">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Profile"
        android:textAlignment="center"
        android:textSize="28sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="24dp"
        android:text="Name"
        android:textSize="16sp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView" />

    <EditText
        android:id="@+id/editTextName"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="24dp"
        android:inputType="textPersonName"
        android:text="Name"
        app:layout_constraintBottom_toBottomOf="@+id/textView1"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/textView1"
        app:layout_constraintTop_toTopOf="@+id/textView1" />


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

CoordinatorLayout

CoordinatorLayout is introduced as part of the Android Design Support Library with Android 5.0 It is designed specifically for coordinating the appearance and behavior of the app bar across the top of an application screen with other view elements. When creating a new activity using the Blank Activity template, the parent view in the main layout will be implemented using a CoordinatorLayout instance.

The Layout XML File vs The Activity Code

In most cases, we define the layout in the XML layout file using either the visual design editor or the text editor. Then in the activity Java code, the layout is loaded and inflated using the `setContentView()` method. Using this approach, all View elements are defined **statically**. You can also instantiate widgets or layout elements at runtime in Java code. The following code shows how to dynamically add a linear layout and set its property.

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState);
   // creating LinearLayout
   val linearLayout = LinearLayout(this);
   / specifying vertical orientation
   linearLayout.orientation = LinearLayout.VERTICAL
   // creating LayoutParams
   val linearLayoutParam = LinearLayout.LayoutParams(
       LinearLayout.LayoutParams.MATCH_PARENT,
       LinearLayout.LayoutParams.MATCH_PARENT)
   // set LinearLayout as a root element of the screen
   setContentView(linearLayout, linearLayoutParam);
}
```

Test Yourself

What design patterns are used in designing Android View and ViewGroup classes?

composite pattern

Test Yourself

What is a possible issue when widgets are not properly positioned and overlapped with each other using the ConstraintLayout?

Missing either horizontal or vertical constraints of the widget

Test Yourself

Which method is used in onCreate() to set the root layout of a screen?

setContentView()

Test Yourself

What layout should be used if you want to stack the components on top of each other?

FrameLayout

Test Yourself

Explore different layouts and widgets by yourself. You can first use the design mode to drag and drop the widgets into the design window. Then check the generated XML file.

Test Yourself

Explore the constraint layout. Re-create activity_main.xml of the Widgets Explore project using the constraint layout instead of linear layout.

Event Listeners and Callbacks

So far, our WidgetsExplore app has a nice UI that can display widgets on the screen, but it cannot react to the user's interaction yet. When the user clicks the submit button, it does nothing. Suppose we want to have a message popped out to tell the user that the submission is successful when the user clicks the submit button. The following sections show how to achieve this.

Toast

A toast is a simple pop-up message displayed on the screen. It will automatically disappear when it times out. Let us create a method named `onClickSubmit()` in the MainActivity class as follows:

```
fun onClickSubmit(){
   Toast.makeText(this, "Submit successfully!", Toast.LENGTH_LONG).show();
}
```

Make sure to **call the show()** method at the end after constructing the toast message. It is a common mistake that the show() method is not invoked. This will pop up a message showing "Submit successfully!" for a long period (about 5 seconds). The question is how to let this method be called when the user clicks the button. There are several ways to hook this code.

The onClick Property

The simplest way is to use the onClick property of the submit button. Modify the button portion in the layout XML file as follows:

```
<Button
   android:id="@+id/buttonId_submit"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="submit"
   …
   android:onClick="onClickSubmit"/>
```

However, this is still not completely correct. You also need to change the above method to include a View parameter as follows:

```
fun onClickSubmit(){
   Toast.makeText(this, "Submit successfully!", Toast.LENGTH_LONG).show();
}
```

Register an OnClickListener

Another way is to register the button with an OnClickListener. An OnClickListener is an interface of the View class that contains a single call back method `onClick()`. We can first create an OnClickListener object which implements the `onClick()` method, then use the method `setOnClickListener()` to hook with the button. We can simply use the following code:

```
// set the submit button event listener
findViewById<Button>(R.id.buttonId_submit)?.setOnClickListener { onClickSubmit() }
```

First, let us understand the setOnClickListener()method, which is defined here View setOnClickListener as following:

```
open fun setOnClickListener(l: View.OnClickListener?): Unit
```

We need to provide a `View.OnClickListener` object as the parameter. The `View.OnClickListener` is an interface defined at View.OnClickListener. It is a functional interface or a *Single Abstract Method (SAM)* interface which is **an interface that contains only one abstract method**. In this case, this abstract method is `onClick(View v)`.

To get a `OnClickListener` object, we will need to first create a class that implements this functional interface by overriding the `onClick(View v)`, then create an object of that class.

```
findViewById<Button>(R.id.buttonId_submit)?.setOnClickListener(
                object: View.OnClickListener{
                        override fun onClick(v: View?) {
                                onClickSubmit()
        }
   })
```

But this is cumbersome. Instead, we can simply use a lambda expression. Kotlin allows some optimisations over Java libraries, and any function that receives an interface with a single abstract function can be substituted by a lambda. In Kotlin {} are always a lambda expression or a part of a syntax construct. Here we use { } for a lambda expression. So it can be simplified to:

```
findViewById<Button>(R.id.buttonId_submit)?.setOnClickListener(
        {v-> onClickSubmit()}
)
```

Since the input v is not used in onClickSubmit(), we can omit it. Also, if the last argument of a function is also a function, we can move it out of the parentheses as simply as:

```
findViewById<Button>(R.id.buttonId_submit)?.setOnClickListener { onClickSubmit() }
```

Android Event Handling

The above example shows how to react to a single click event on the button. Now, let us explore a little bit more about Android event handling. When a user interacts with an app such as clicking, swiping, or pressing, he/she essentially interacts with a specific View object, e.g. a button, a text field, or even the whole screen area (the layout). To handle an event, Android needs to be able to capture that event from a specific View. To capture the event, the View must register the appropriate event listener and implement the corresponding callback. For example, in the above example, to capture the onClick event on the button, the button needs to register onClickListener which implements the onClick() method. When an event happens, Android will capture that event, put the event into the event queue, usually in the FIFO (First In First Out) order, pass the event to the View where the event took place, along with other information about the event (e.g. coordinates, type, etc), and then calls the implemented callback function.

Event Listeners and Callbacks

An eventListener is an interface with a single callback function. Android supports several types of event listeners and callbacks. A View registers an event listener using a setter method. For example, setOnClickListener() is used to register OnClickListener,

setOnKeyListener() is used to register OnKeyListener.

EventListener	Callback
OnClickListener	onClick(v: View): Unit
OnLongClickListener	onLongClick(v: View): Boolean returns a Boolean to indicate if the event was consumed If it returns true, the event is discarded by the Android framework. If it returns false, the event is still active and is passed along to the next matching event listener.
OnTouchListener	onTouch(v: View, m: MotionEvent): Boolean The MotionEvent object is passed to this callback. It is the key to obtain the information about the event such as the location of the touch and action type (e.g. ACTION_DOWN. ACTION_UP, ACTION_MOVE)
OnCreateContextMenuListener	onCreateContextMenu(menu: ContextMenu menu, v: View, menuInfo: ContextMenu.ContextMenuInfo):Unit
OnFocusChangeListener	onFocusChange(v: View v,hasFocus: Boolean): Unit
OnKeyListener	onKey(v: View, keyCode: Int, event: keyEvent): Boolean

Other Event Handlers

Besides the above event listeners, there are several other event handlers defined in the Keyevent.Callback interface. Both Activity and View class implement this interface. You can directly override these event handlers in your Activity class, or create a customized View and override them.

`onKeyDown(int, KeyEvent): Boolean` - Called when a new key event occurs
`onKeyUp(int, KeyEvent): Boolean` - Called when a key up event occurs
`onTrackballEvent(MotionEvent): Boolean` Called when a trackball motion event occurs
`onTouchEvent(MotionEvent): Boolean` Called when a touch screen motion event occurs
`onFocusChanged(boolean, int, Rect):` Boolean - Called when the view gains or loses focus

Complete the WidgetsExplore Project

Getting the Information from Widgets

Though it is nice to pop up a “Submit successful” message when clicking on the submit button, it is not very useful. It is supposed to get all information entered by the user and then stored in the database or sent to the server. We will discuss the database and remote connections in the later modules. Let us first figure out how to get the information the user entered or selected in each

widget. First, we need get a reference to the widget object based on its id defined in the XML file using the findViewById() method, then we need to get its attribute directly or call the corresponding getter method to get the information of that widget, such as `getText()`, `getSelectedItem()`, `getCheckedRadioButtonId()`, etc. The following code shows how to implement this. After the user submits the info, we also clear all fields.

```
private fun onClickSubmit() {
   // get the name from the EditText
   val name = nameET.text.toString().trim()
   val country = countrySp.selectedItem.toString()
   val isAdult = isAdultCb.isChecked
   val isAdultStr = if(isAdult) "18 years or older " else "18 years younger"

   // get the gender from the RadioGroup
   val gender = when (genderRG.checkedRadioButtonId) {
       R.id.radioButtonId_female -> "female"
       R.id.radioButtonId_male -> "male"
       else -> "unknown"
   }

   // get isPublic from the Switch
  val isPublic = isPublicSw.isChecked
  val isPublicStr= if (isPublic) "public" else "private"

   // get comments from the Multiline EditText
   val comments = commentsET.text.toString()

   // compose a message from the above information
   if (name.isNotEmpty()) {
       findViewById<TextView>(R.id.textViewId_info).text =
           getString(R.string.yourinfosummary, name, country, gender,
               isAdultStr, isPublicStr, comments)
       // display a toast message
       Toast.makeText(this, "Submit successfully!", Toast.LENGTH_LONG).show()
   } else
       Toast.makeText(this, "Please input your name!", Toast.LENGTH_LONG).show()

   clearData()

}
```

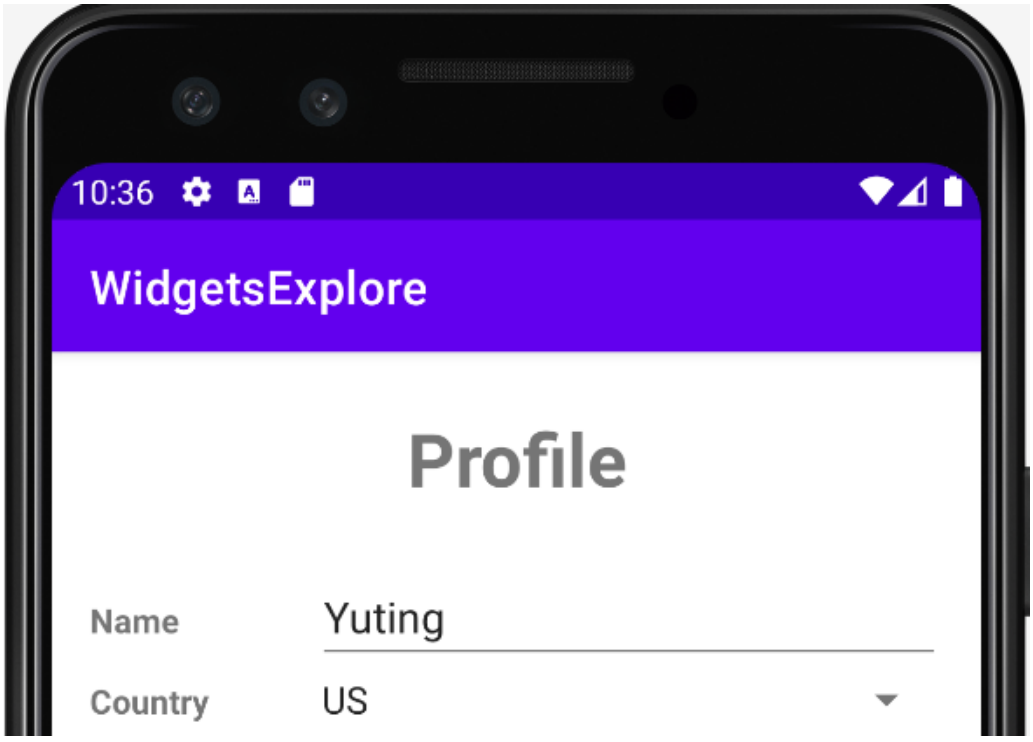
Here we define the string template in the resource file strings.xml.

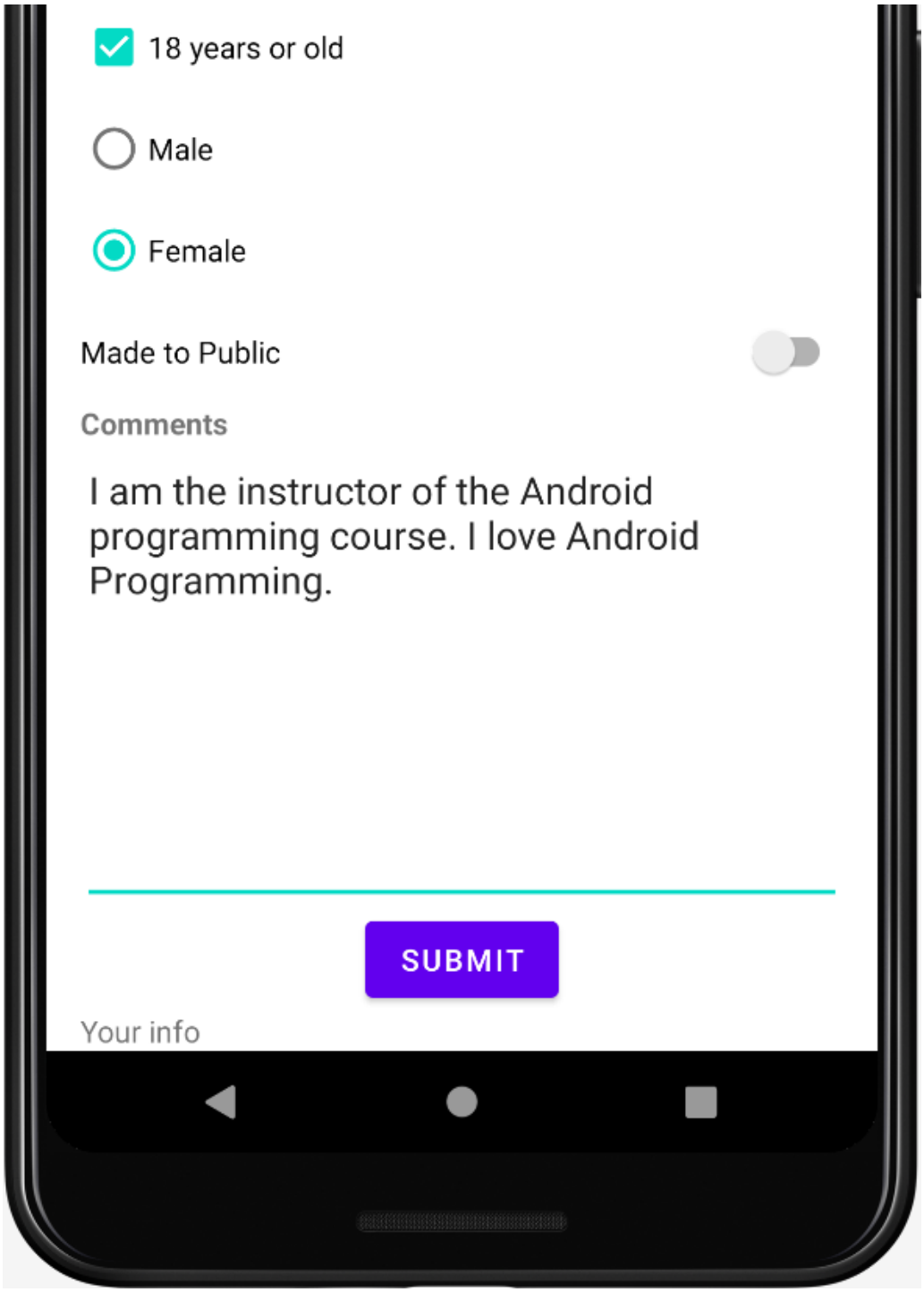
```
<string name="yourinfosummary">
   Here is your information:\n
   %1$s in %2$s\n
   %3$s (%4$s)\n
   Your info is made %5$s\n
   Your comments: %6$s</string>
```

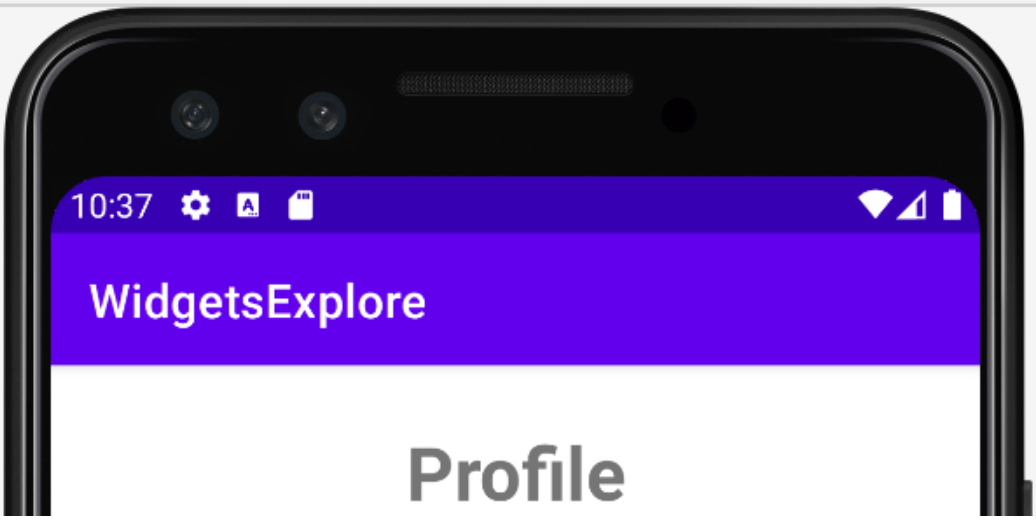
We can also use raw strings without using the escape “\n” to compose the message directly in the Kotlin code as:

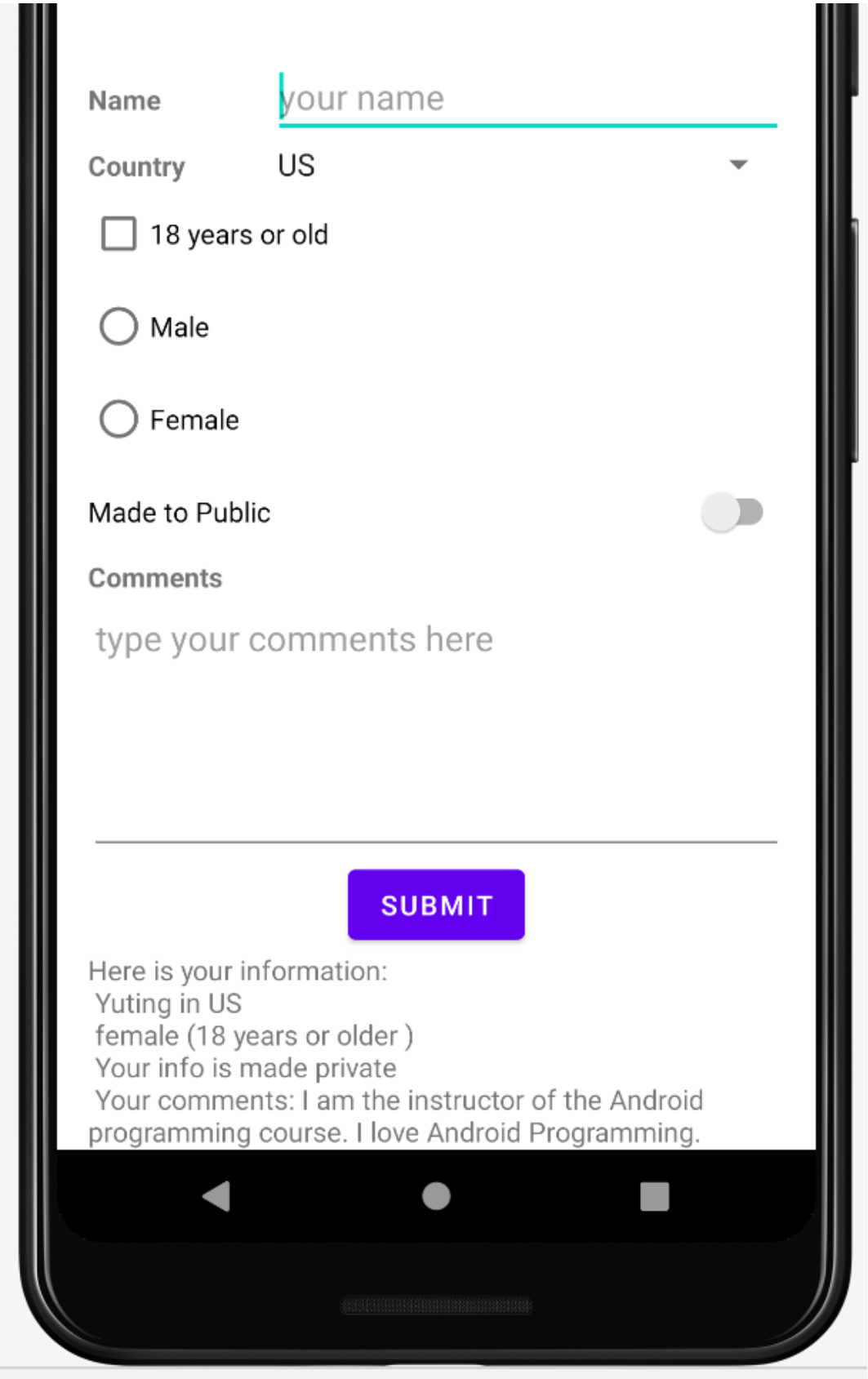
```
val msg = """
   Here is your information:
   $name in $country
   $age
   $gender
   Your info is made $public
   Your comment:
   $comments
   """.trimIndent()
```

The execution results are shown as follows.




18 years or old
Male
Female
Made to Public
Comments
I am the instructor of the Android programming course. I love Android Programming.
SUBMIT
Your info


10:37
WidgetsExplore
Profile



Test Yourself

What is a Toast message? Write a line of code to display a short Toast message “Hello”. Why can you call the makeText() method directly without instantiating a Toast object? What does this method return?

A Toast message is a pop up message shown on the screen for a time period (usually a couple of seconds). Because makeText() is a static method of the Toast class. It returns a Toast object.

```
Toast.makeText(this, "hello", Toast.LENGTH_LONG).show();
```

Test Yourself

What is an event listener in Android? What method is used to register an event listener to a view? What events does Android support? Please list a few.

An event listener in Android is an interface with a single callback function to handle the event. A set event listener method of the View object is used to register the event listener for each event. For example, setOnClickListener() is used to register onClickListener to handle the event onClick. Call button.setOnClickListener() registers the onClickListener to the button object. Supported events include: onClick, onTouch, onLongClick, onKeyListener, etc)

Test Yourself

One common parameter in all event handlers e.g. onClick(), onTouch() is _____.

a View

Test Yourself

Which method is used to get the result of the selected item in Spinner.

getSelectedItem()

Test Yourself

Explain how to statically bind a button onClick event with a method defined in the activity.

The onClick attribute of the Button in the XML file.

Test Yourself

Implement the clearData() method called in onClickSubmit() to:

a. reset all fields. (use set methods of each widget)
b. Move the cursor back to the name EditText. (Use requestFocus())

adding following lines at the end of onClick()) method.

```
nameET.text?.clear()
countrySp.id = 0
isAdultCb.isChecked = false
```

```
        genderRG.clearCheck()
        isPublicSw.isChecked = false
        commentsET.text?.clear()
        nameET.requestFocus()
```

Topic 2: Inter Component Communication and UI Navigation

The Project Portal Example

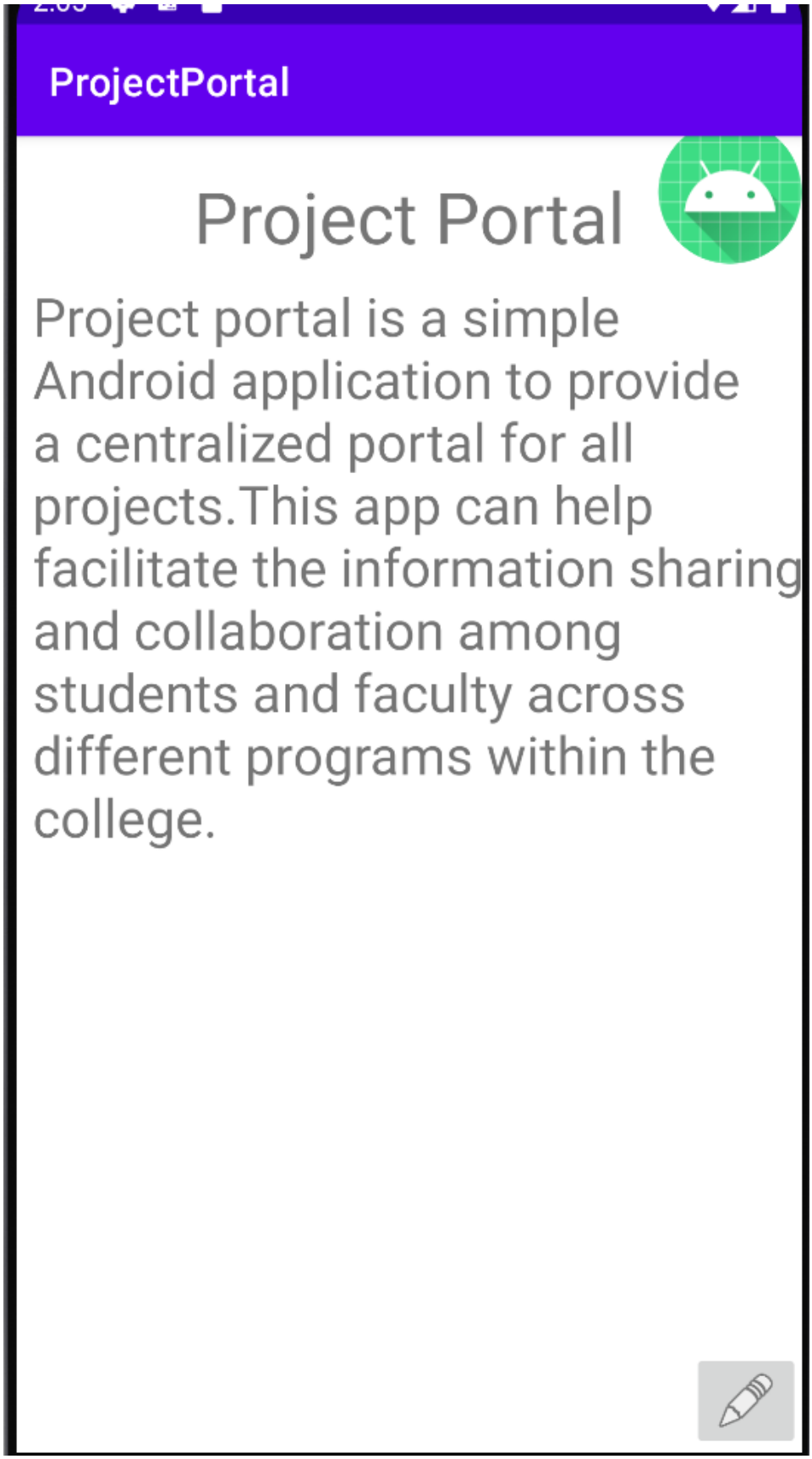
In the previous example, we only created one screen. In most applications, we will need multiple screens and the user can navigate between them.

Let us use the ProjectPortal app as an example. Suppose we want to view a project detail as well as edit the project information. We will create two screens, one is to simply display the project details, and another is to edit the project info. For now, we can create two different activities, each with its own XML layout file.

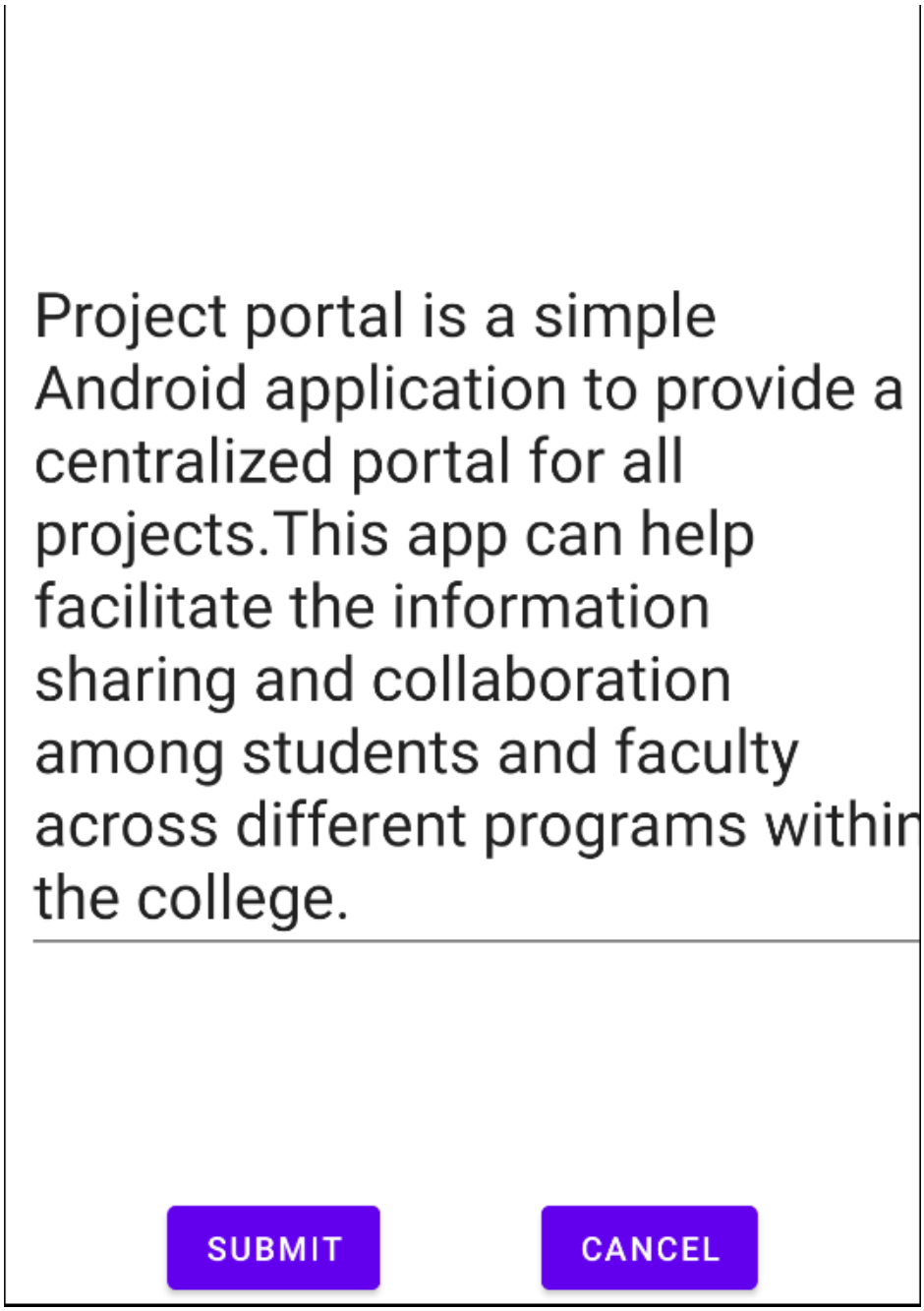
To separate the data from its presentation, we will first create a data class for Project. For now, we only have a title and description for each project. For simplicity we also pre-define a project “Project Portal” using a companion object. In Kotlin, there are no “static” variables. However, we can use a companion object to achieve the similar result. We can simply call it through the class name.

```
data class Project(val id: Int, var title: String, var description: String){
   companion object {
       val project = Project(0, "Project Portal",
           "Project portal is a simple Android application to  " +
                   "provide a centralized portal for all projects." +
                   "This app can help facilitate the information and " +
                   "sharing and collaboration among students and faculty " +
                   "across different programs within the college.")
   }
```

We will need two screens: one screen to display all projects, and another to display the detail. We can create two activities: ProjectDetailActivity and EditProjectActivity. On the ProjectDetail screen, we use two TextViews for title and description, an image for project icon, and an image button to trigger the edit event. On the EditProject screen, we use two EditText for title and description, an image for project icon, and two basic buttons to navigate back to the detail screen. Students can try to create these XML layout files by yourselves.



Project Portal



When the user clicks on the edit image button on the detail screen, the edit screen will be displayed for users to edit title or description. Once it is done, the user can click on the submit or cancel button. It will navigate back to the detail page, reflecting new changes. So how to implement this navigation?

Intent and ICC

An Intent is a messaging object you can send from one android component to another. Whenever we want to navigate from one activity to another, we can use Intent. In the traditional OOP (Object Oriented Programming), when an object obj1 calls obj2.method(parameters), obj1 basically sends a message to obj2. However, this communication is more static, and requires the obj2 name and its method name known in advance. Instead of using method calls directly, Android uses an intent message for component communication that encapsulates the receiving component, action(method) and data(parameters) together in a single object.

In Module 1, we introduced 4 different Android components: activity, service, broadcast receiver and content provider. An Android component such as an activity, a service, or a broadcast receiver can send an intent to another component. However, the content provider cannot send or receive intents though they are also android components. There are several different methods used to send an intent depending on the receiving component type. They are defined in the `Context` class. The `Context` class in Android is the interface to global information about an application environment. Here are several methods to send an intent message.

- Start an activity: `startActivity(intent)`
- Start a service: `startService(intent), bindService(intent)`
- Send a broadcast intent: `sendBroadcast(intent)`

We will discuss services and broadcast receivers in the later modules. In this module, we only discuss ICC between activities. To start an activity, we need to first create an intent, and then call startActivity(intent). Since the activity class is an indirect subclass of the Context class, we can call startActivity() directly in an activity. However, if we want to call this method in other classes, we may need to get a reference to the current context first. Several methods can be used to return the current context references:

- View.getContext(): Returns the context the view is currently running in. Usually the currently active Activity.
- fragment.getActivity(): Return the FragmentActivity this fragment is currently associated with.
- Activity.getApplicationContext(): Returns the context for the entire application (the process all the Activities are running inside of). Use this instead of the current Activity context if you need a context tied to the lifecycle of the entire application, not just the current Activity.
- ContextWrapper.getBaseContext(): If you need to access a Context from within another context, you use a ContextWrapper. The Context referred to from inside that ContextWrapper is accessed via getBaseContext().

Several constructor methods can be used to create an intent. The most common one is to specify two parameters: the sending component, and the receiving component.

Java:

```
Intent intent = new Intent (this, Target.class)
```

Kotlin:

```
val intent = Intent(this, Target::class.java)
```

The first parameter `this` means the sending component is the current activity. The second parameter `Target::class` specifies the receiving component defined by the `Target` class. This is through reflection, a feature in Java.

In addition to specifying the receiving activity, we can also include additional information in the intent, such as the action name, data, flags, or extra information. This can be done through various constructor methods.

Intent()	Create an empty intent.
Intent(o: Intent!)	Copy constructor.
Intent(**action:** String!)	Create an intent with a given action.
Intent(**action:** String!, uri: Uri!)	Create an intent with a given action and for a given data url.

Intent(**packageContext**: Context!, cls: Class<*>!)	Create an intent for a specific component.
Intent(action: String!, uri: Uri!, packageContext: Context!, cls: Class<*>!)	Create an intent for a specific component with a specified action and data.

You can also directly access/modify the intent data in Kotlin or using a number of explicit getter and setter methods in Java.

- `setAction(), setClass(), setComponent(), setData(), setFlags(), putExtra()...`
- `getAction(), getComponent(), getData(), getFlags(), getExtras(), …`

For more details, please see: Intent.

To access the received intent, we can access that intent directly or call getIntent() (in Java) in the receiving activity, and get the corresponding information included in the intent.

Here is the simple code snippet to navigate from the ProjectDetail Activity to the EditProject Activity.

```
editProj.setOnClickListener {
   startActivity(Intent(this, EditProjectActivity::class.java))
}
```

Here is the simple code snippet to navigate from the EditProject Activity back to the ProjectDetail Activity. Since this navigation will happen no matter the user clicks on the submit or cancel button, here we define a single OnClickListener and set it to both submit and cancel button. However, the project information is only updated when the submit button is clicked.

```
val editProjDoneListener = View.OnClickListener{ view ->
   if (view.id == R.id.submit) {
       Project.project.title = projTitle.text.toString()
       Project.project. description = projDesc.text.toString()
   }
   startActivity(Intent(this, ProjectDetailActivity::class.java))
}

submit.setOnClickListener (editProjDoneListener)
cancel.setOnClickListener (editProjDoneListener)
```

Topic 3: Fragments

Fragments vs. Activities

An activity is usually mapped to a single screen. It is an independent Android component. Even if you want another similar screen, you cannot reuse the previous activity. You have to create another activity. If you want to split up the same screen or combine multiple screens in a different sized mobile device, you have to create new activities too.

Fragments are introduced in Android 3.0 to address this issue. (If the targeted device is below API 11(Android 3.0), then you need Androidx to provide backwards compatibility). It provides more reusability, modularity and flexibility in the UI designs, particularly making it more adaptable for different screen sizes. In the Android development website (Fragments), they give the following example. Consider an app that responds to various screen sizes. On larger screens, the app should display a static navigation drawer and a list in a grid layout. On smaller screens, the app should display a bottom navigation bar and a list in a linear layout. Since the list displayed should be the same for different screen sizes, we would like to reuse it. We can use a fragment to display the list with proper layout, while the activity is responsible for displaying the correct navigation UI.

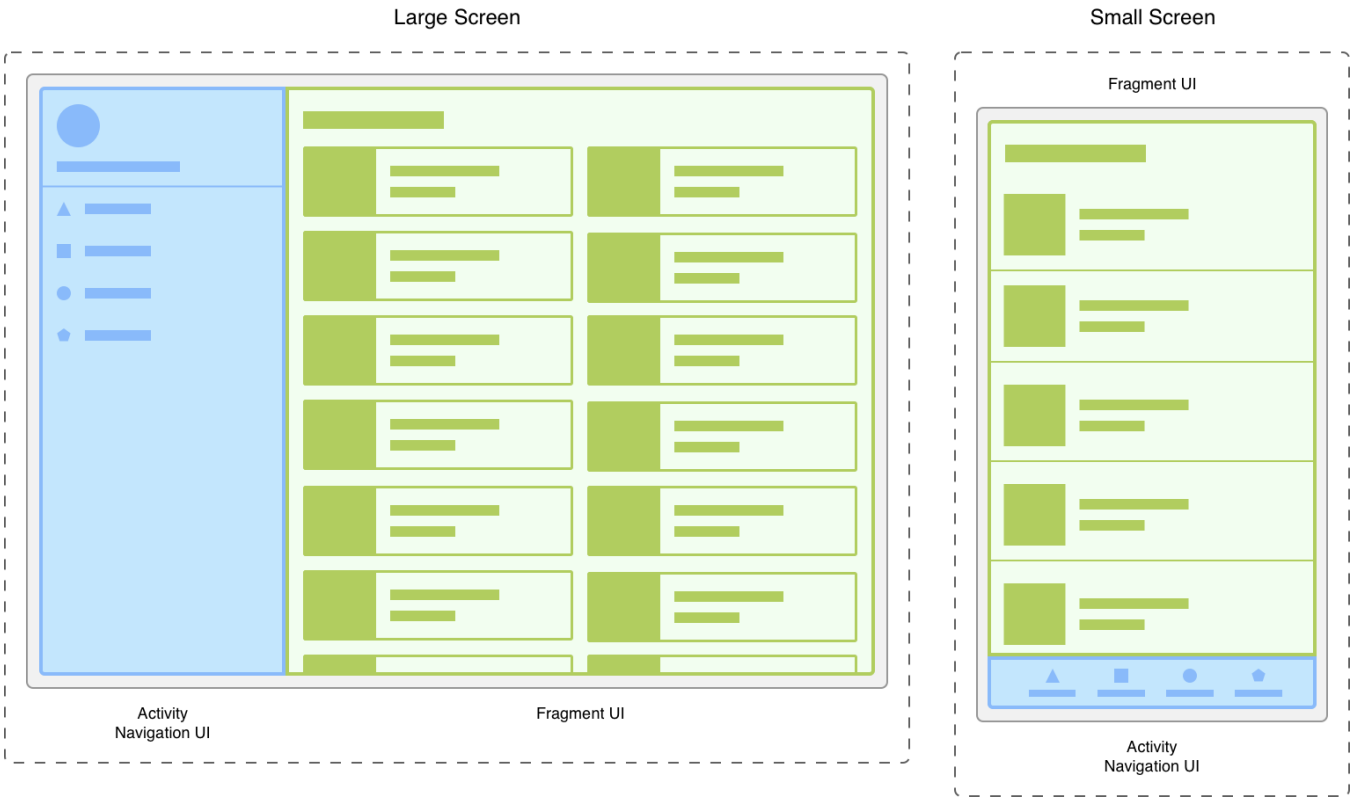


Figure 2.1. Two versions of the same screen on different screen sizes. On the left, a large screen contains a navigation drawer that is controlled by the activity and a grid list that is controlled by the fragment. On the right, a small screen contains a bottom navigation bar that is controlled by the activity and a linear list that is controlled by the fragment.

Fragments: Definition and Features

A fragment is a self-contained, modular section of an application's user interface and corresponding behavior that can be embedded within an activity. It is sort of like a "sub activity". Multiple fragments can be combined in a single activity to build a multi-pane UI. A fragment has the following features:

- It usually has a layout and receives its own input events. (Similar to an activity)
- It has its own lifecycle. (Similar to an activity)
- It can be added or removed to create a dynamically changing user interface at runtime.
- It can be reused in multiple activities.
- Multiple fragments can be combined in a single activity.

- It **cannot** be instantiated as a standalone application element and can only be used as part of an activity. (Different from an activity)

> ### Test Yourself
> Comparing activities and fragments, what are similarities and differences? What is their relationship?
>
> Similarities: Both usually have a layout and can handle input events. Both have life cycles.
> Differences: An Activity is the entry point of a UI screen. It is defined in the manifest file. It can be used alone. A fragment cannot be used alone, and is not defined in the manifest file. It can only be used within some activity.
> Relationship: An activity can have several fragments and a fragment can be used in several activities.

How to Use Fragments in Your Project

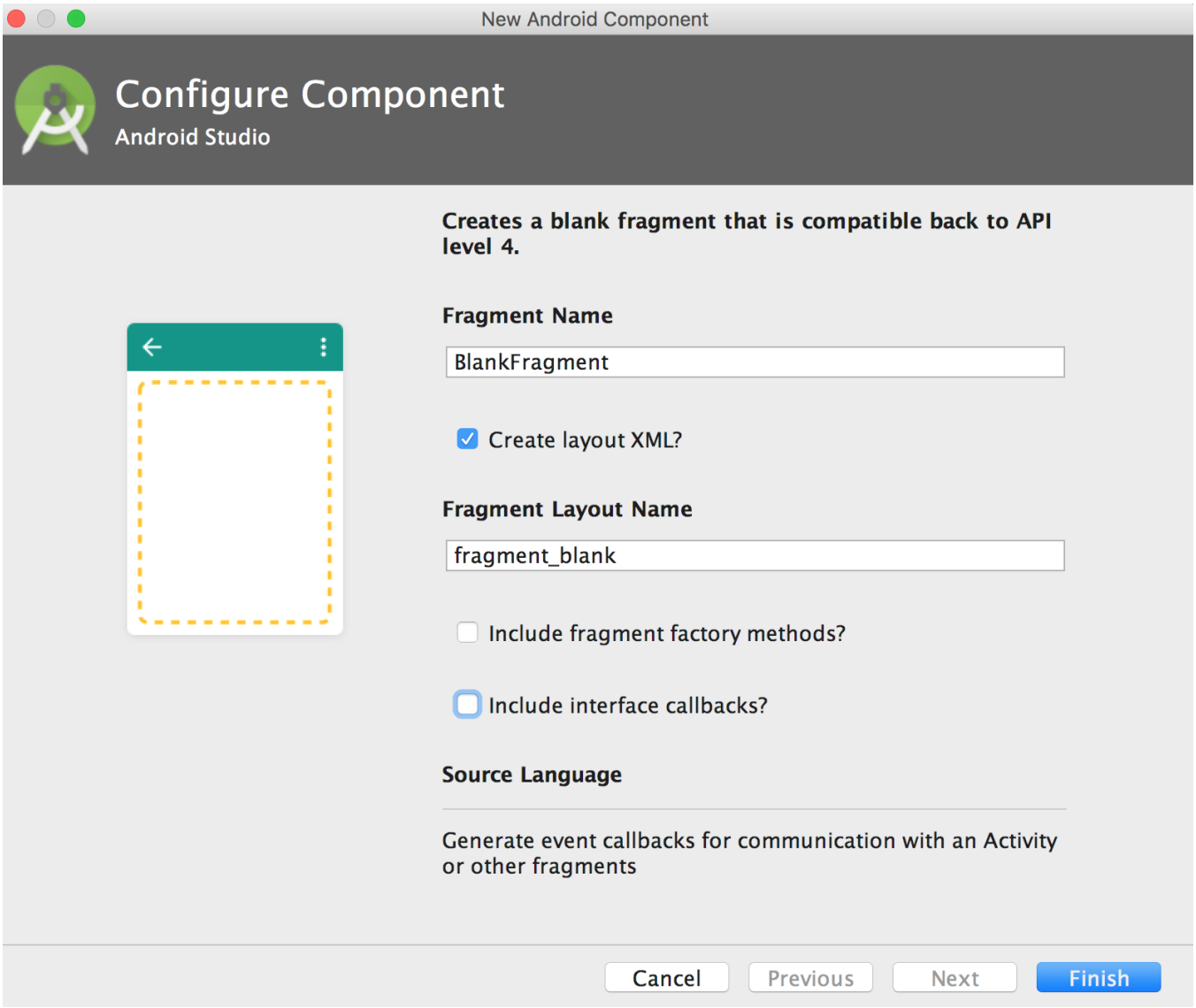
Now, let us talk about how to use fragments in your project. We will use the ProjectPortal app as an example. Suppose we want to display a list of projects done in this course. When clicked on a single project, the details of this project are displayed. So we need to create two fragments: ProjectListFragment and ProjectDetailFragment.

The Fragment Class

The Fragment class was originally defined in the app package in the Android framework as following (Nested Classes). But this class was deprecated in API level 28, and it is recommended to use the JetPack Fragment Library `androidx.fragment.app.Fragment` for consistent behavior across all devices and access to Lifecycle.

Create Fragments

Similar to an activity, a fragment is usually associated with two files: a java/Kotlin code to define a fragment class and a XML file for its layout. The Java/Kotlin code defines its behavior (e.g. how to respond to a button click), and the XML describes its initial appearance. Android Studio will create two files for you when you create a fragment. You can also choose a template for your fragment. Here we choose the Blank fragment. You may also uncheck “Include fragment factory methods” and “Include interface callbacks” to minimize the code generated. You shall also change the name to the desired name.



By default, Android Studio creates the Frame layout as the root level layout. However, you can change this layout or add additional layouts as needed. In the Java/Kotlin code, you need to subclass the Fragment class and override some callback methods. By default, Android Studio uses the Fragment class from the JetPack library now for backward compatibility.

One of the important methods to override is the `onCreateView()` method which specifies the layout the fragment uses if a layout is used for the fragment. The `LayoutInflater` object is used to inflate the fragment's layout which turns the XML views into View objects. The `container` parameter of the `inflate()` method also needs to specify where (which ViewGroup) the fragment should be inflated. This is the fragment equivalent of calling an activity's `setContentView()` method. You can also implement other callback methods such as onCreate(), onStart(), onPause(), and onStop() as needed.

Suppose we create a XML layout fragment_project_detail.xml using the ConstraintLayout as follows.

```
<?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/detailFragment"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".DetailFragment">

<TextView
    android:id="@+id/projTitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="Project Title"
    android:textSize="16pt"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="@+id/projTitle"
    android:src="@mipmap/ic_launcher"
    tools:src="@tools:sample/avatars" />

<TextView
    android:id="@+id/projDesc"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:layout_marginStart="32dp"
    android:layout_marginEnd="16dp"
    android:textSize="12pt"
    android:text="Project Description"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/imageView" />


<ImageButton
```

```
        android:id="@+id/editProj"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@android:drawable/ic_menu_edit"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toBottomOf="parent" />

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

The corresponding Fragment Kotlin code is shown as below. The method onCreateView() is overridden to specify the layout file.

```
class DetailFragment : Fragment() {
   override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
   ): View {
        return inflater.inflate(R.layout.fragment_detail, container, false)
   }
}
```

If this fragment only needs to define its own layout, we can even provide this layout resource to the base constructor directly as:

```
class DetailFragment: Fragment(R.layout.fragment_detail)
```

Of course, in this fragment, we need to do more things. We will override and add other methods

Add Fragments

Since a fragment **CANNOT** be used as a standalone component, we need to create an activity and add the fragment into it. Let us create an empty activity ProjectDetailActivity, and add the fragment into it. We can add the fragment statically through the XML layout file. We can also add it dynamically in the Java/Kotlin code. For example, if the fragment is the only component in this activity, we can even add the fragment into the activity layout file as the root level component as below. It is strongly recommended to always use a `FragmentContainerView` as the container for fragments.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/container"
    android:name="edu.bu.projectportal.DetailFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" />
```

We can also add the fragment within other Layouts. If the activity also contains other components besides the fragment, we will need a root level layout such as ConstraintLayout, and put all components into it.

When a fragment is added into the xml file, it is static. Sometimes, we may need to dynamically add or remove or replace fragments through Java/Kotlin code. To add fragments dynamically, you need to use FragmentManager. For compatibility, we shall use `SupportFragmentManager` defined in AndroidX. For example, here we can replace the fragment that insides R.id.container to a DetailFragment in MainActivity.

```
override fun editProj(){
    supportFragmentManager.commit{
        replace<EditFragment>(R.id.container)
    }
}
```

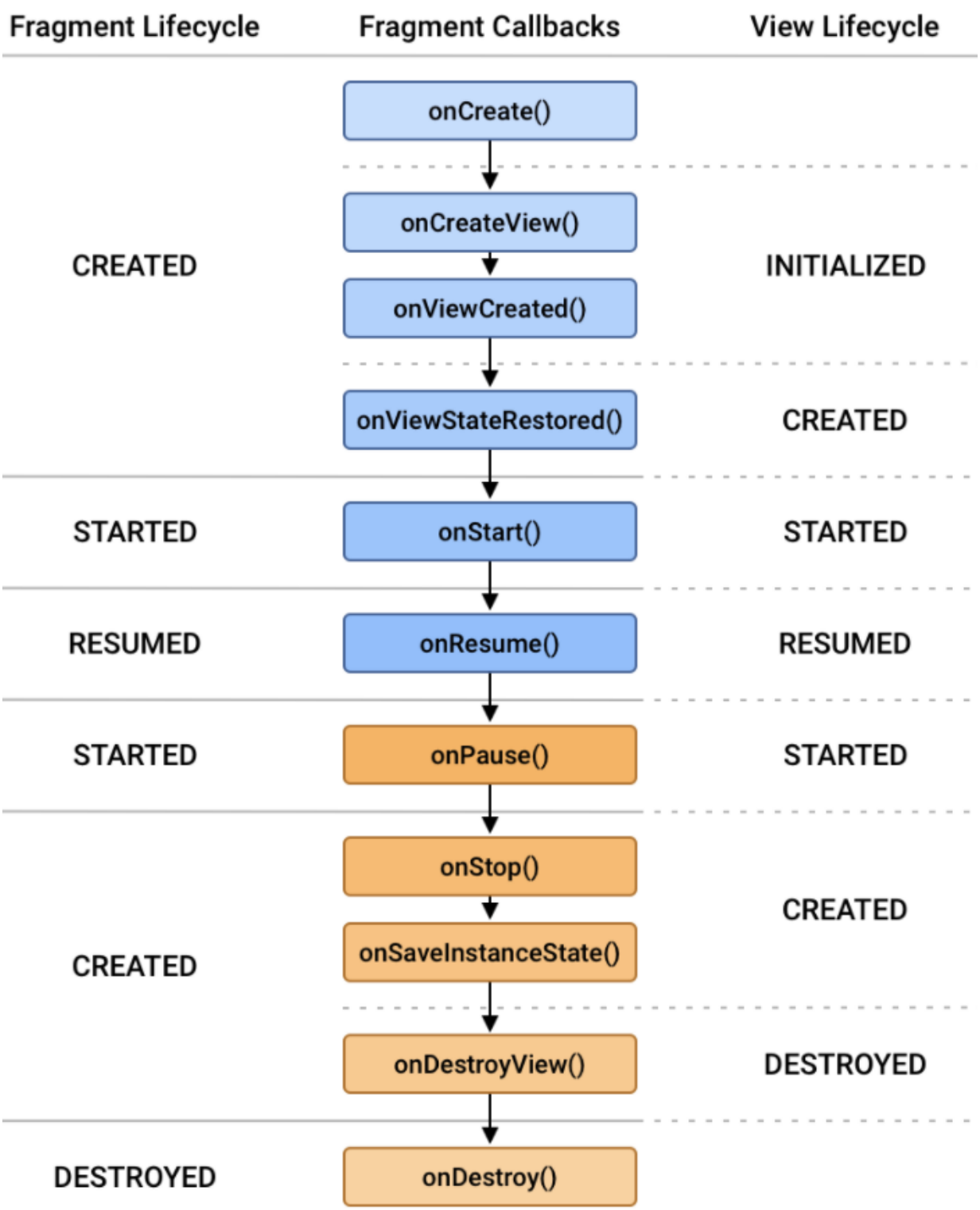
The supportFragmentManager.commit{...} is defined in fragment-ktx. We need to add the following in the module gradle file

```
implementation "androidx.fragment:fragment-ktx:1.2.5"
```

You can also remove a fragment previously added, or replace a fragment with a different fragment, or hide and show an existing fragment using supportFragmentManager. For more details, please refer to Fragment Manager.

The Fragment Lifecycle

A fragment has its own lifecycle. It starts in the INITIALIZED state when it is instantiated, and it transitions through different states when it is added, when it is added, or removed or replaced by a FragmentManager. The FragmentManager also attaches/detaches it to the host activity. A fragment can be added to or removed from the back stack. It can also return to the layout from the back stack (e.g. navigate backwards by pressing the Back button). The following diagram shows the fragment lifecycle and its mapping to fragment callback as well as the fragment's view lifecycle.



Be aware that a fragment is not a standalone component, and thus cannot be used by itself. Unlike an activity, it is not a type of Context. Instead, a fragment must be used within some activity and accessing the application environment using the context of its host activity.

Handling Fragment Events

The view components within a fragment can generate events just like those in a regular activity. Whether the fragment or the activity should handle such events depends on how the event handler is declared.

The general rule for events generated by a view in a fragment is that if the event listener was declared in the fragment class using the event listener and the callback method approach, then the event will be handled first by the fragment. But if the android:onClick resource is used, the event will be passed directly to the activity containing the fragment.

Navigation

We can use the JetPack Navigation Component to simplify the communication among fragments/activities. There are three parts in the JetPack Navigation component:

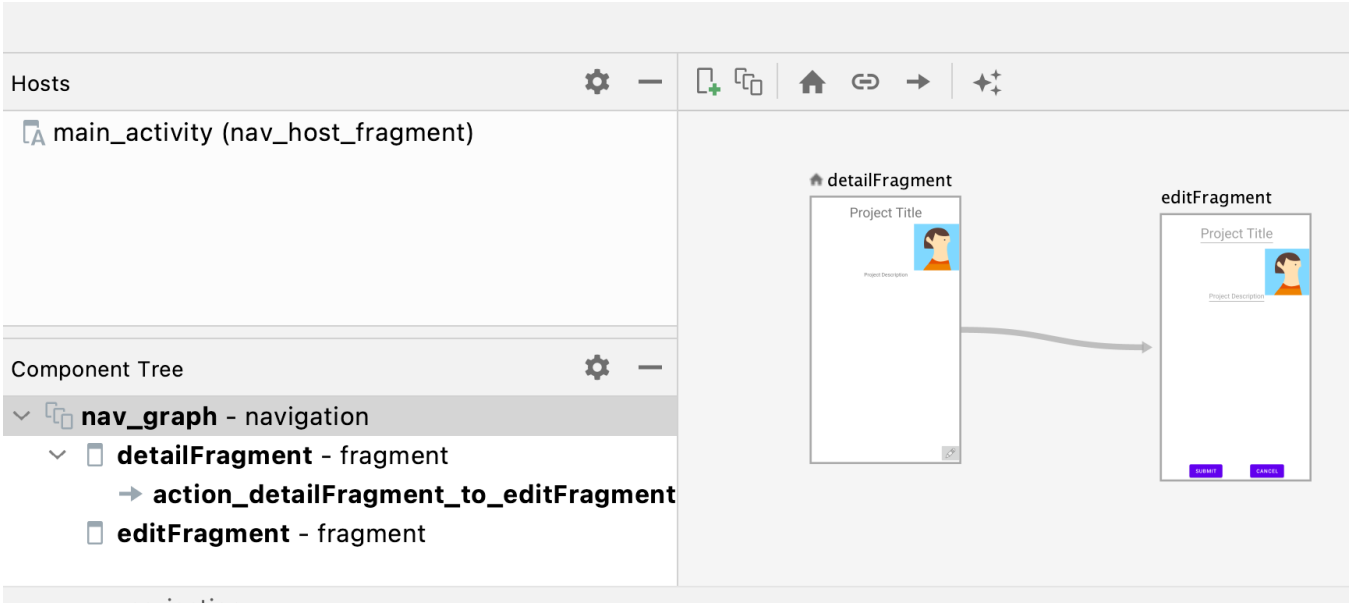
- Navigation graph: an XML resource that contains all navigation-related information.
- NavHost: An empty container to display destinations from your navigation graph. The Navigation component contains a default NavHost implementation, NavHostFragment, that displays fragment destinations.
- NavController: An object that manages app navigation within a NavHost.

Create a Navigation Graph

The navigation graph is defined as a resource file (res/navigation/nav_graph.xml). One can use the “design” mode to drag and drop activities/fragments into the graph. In a navigation graph, we need to:

- Define a fixed start destination, which is also the first and last screen.
- Add other destinations and define the screen navigation and actions.

In a fragment Back stack, the top one is the current screen, and the bottom one is the start destination. After navigating to the destination screen, we can go back to the current screen by pressing the back button as well which simply pop up the destination from the back stack.




```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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/nav_graph"
    app:startDestination="@id/detailFragment">

    <fragment
        android:id="@+id/detailFragment"
        android:name="edu.bu.projectportal.DetailFragment"
        android:label="fragment_detail"
        tools:layout="@layout/fragment_detail" >
        <action
            android:id="@+id/action_detailFragment_to_editFragment"
            app:destination="@id/editFragment" />
    </fragment>
    <fragment
        android:id="@+id/editFragment"
        android:name="edu.bu.projectportal.EditFragment"
        android:label="fragment_edit"
        tools:layout="@layout/fragment_edit" />
</navigation>
```

Add a NavHost to an Activity

We need a NavHost as the container for destinations. The NavHostFragment is a default NavHost implementation. It can handle swapping fragment destinations. Add this into the layout XML file for the activity, and specify the navigation graph.

```
<androidx.fragment.app.FragmentContainerView
    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/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/nav_graph"
    tools:context=".MainActivity"/>
```

Navigate to a Destination Using NavController

For example, we need to navigate to the EditFragment when the edit image button is clicked. We need to first get the reference to the NavController. This is done by calling findNavController() on the root view object of the fragment.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view,savedInstanceState)
        editProj.setOnClickListener{
            view.findNavController().
            navigate(R.id.action_detailFragment_to_editFragment)
        }
    }
}
```

When navigating fragments, Android maintains a back stack that contains the destinations you've visited. The first destination of your app is placed on the stack when the user opens the app. Each call to the navigate() method puts another destination on top of the stack. Tapping Up or Back calls the NavController.navigateUp() and NavController.popBackStack() methods, respectively, to remove (or pop) the top destination off of the stack. You can find more details at Navigation.

Test Yourself

Which class is an indirect subclass of the Context class?

Activity

Fragment

Test Yourself

What is the difference between android.app.Fragment and androidx.fragment.app.Fragement?

The latter one provides backward compatibility with old versions.

Test Yourself

In which callback method, the layout is inflated in the Fragment? Which method is used to inflate the layout?

onCreateView(), inflate() of LayoutInflater

Test Yourself

Write a snippet of code to add a fragment dynamically into the activity using FragmentManager.

See the ProjectPortal code example in the “Add Fragments” section

Test Yourself

Download the ProjectPortal project, add some log code in every callback method in the activity and fragment. Check the log output when run the application, rotate the device and close the application. Try to understand the lifecycle of an activity and the lifecycle of a fragment.

Test Yourself

What are three parts in the JetPack Navigation component to enable the navigation between fragments?

Navigation graph, NavHost and NavController.

Conclusion

In this module, we introduce several basic Android UI related classes: Activity, View, ViewGroup, and Fragment. We also discuss the relationship between the XML Layout and the Java/kotlin code. An activity is the entry point to UI. An activity can consist of one or multiple fragments. Each activity or fragment will load and inflate an XML layout which defines its initial look. The XML Layout file defines the attributes of the root level layout and View/ViewGroup objects within it. View and ViewGroup objects will be created when the XML layout is inflated. The activity or fragment then can register its view/viewgroup objects to various events and handle them correspondingly. We also discussed how to use intent to navigate between activities and Android JetPack navigation component to enable navigation between fragments.Through two examples: WidgetsExplore and ProjectPortal, we show how to use these classes to create simple Android applications.