MCA253 - Mobile Applications

Unit:2.2
Designing User Interface with Views

Dr. Siddesha S MCA, M.Sc Tech (by Research) , Ph.D
Asst. Professor,
Dept. of Computer Applications,
JSS Science and Technology University
Mysuru – 570 006

ViewGroups

- This part covers following different ViewGroups
  - **Basic views**—Commonly used views, such as the TextView, EditText, and Button Views.
  - **Picker views**—Views that enable users to select from a list, such as the TimePicker and DatePicker views
  - **List views**—Views that display a long list of items, such as the ListView and the SpinnerView views
  - **Specialized fragments**—Special fragments that perform specific functions

Using Basic Views

- Some of the **basic views** that you can use to design the UI of your Android applications:
  - ➢ TextView
  - ➢ EditText
  - ➢ Button
  - ➢ ImageButton
  - ➢ CheckBox
  - ➢ ToggleButton
  - ➢ RadioButton
  - ➢ RadioGroup

TextView View

- When you create a new Android project, Android Studio always creates the activity_main.xml file (located in the res/layout folder), which contains a <TextView> element.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello" />
</LinearLayout>
```

Button, ImageButton, EditText, CheckBox, ToggleButton, RadioButton, and RadioGroup Views

- There are some other basic views that are frequently using:
- Button - Represents a push-button widget.
- ImageButton - Similar to the Button view, except that it also displays an image.
- EditText - A subclass of the TextView view, which allows users to edit its text content.
- CheckBox - A special type of button that has two states: checked or unchecked.
- RadioGroup and RadioButton - The RadioButton has two states: either checked or unchecked. A RadioGroup is used to group one or more RadioButton views, thereby allowing only one RadioButton to be checked within the RadioGroup.
- ToggleButton- Displays checked/unchecked states using a light indicator.

Program: Basicview

Different Views used

- For the first Button, the layout_width attribute is set to fill_parent, which makes its width occupy the entire width of the screen:

```
<Button android:id="@+id/btnSave"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="save" />
```

- For the second Button, the layout_width attribute is set to wrap_content so that its width will be the width of its content—specifically, the text that is displayed (for example, Open):

```
<Button android:id="@+id/btnOpen"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Open" />
```

Different Views used

- The ImageButton displays a button with an image. You set the image through the src attribute. In this code, note that an image has been used for the application icon:

```
<ImageButton android:id="@+id/btnImg1"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:src="@drawable/ic_launcher" />
```

- The EditText view displays a rectangular region in which the user can enter text. In this example, layout_height has been set to wrap_content so that the text entry location automatically adjusts to fit the amount of text entered by the user

```
<EditText android:id="@+id/txtName"
android:layout_width="fill_parent"
android:layout_height="wrap_content" />
```

Different Views used

- The CheckBox displays a check box that users can tap to check or uncheck:

```
<CheckBox android:id="@+id/chkAutosave"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Autosave" />
```

- If you do not like the default look of the CheckBox, you can apply a style attribute so that the check mark is replaced by another image, such as a star:

```
<CheckBox android:id="@+id/star"
style="?android:attr/starStyle"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

- The format for the value of the style attribute is as follows:

```
? [package:] [type:] name
```

Different Views used

- The RadioGroup encloses two RadioButtons. This is important because radio buttons are usually used to present multiple options to the user for selection. When a RadioButton in a RadioGroup is selected, all other RadioButtons are automatically unselected:

```
<RadioGroup android:id="@+id/rdbGp1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >
 <RadioButton android:id="@+id/rdb1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Option 1" />
 <RadioButton android:id="@+id/rdb2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Option 2" />
</RadioGroup>
```

Different Views used

- Notice that the RadioButtons are listed vertically, one on top of another. If you want to list them horizontally, you need to change the orientation attribute to horizontal. You would also need to ensure that the layout_width attribute of the RadioButton views are set to wrap_content:

```
<RadioGroup android:id="@+id/rdbGp1"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:orientation="horizontal" >
<RadioButton android:id="@+id/rdb1"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Option 1" />
<RadioButton android:id="@+id/rdb2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Option 2" />
</RadioGroup>
```

Different Views used

- The ToggleButton displays a rectangular button that users can toggle on and off by clicking:

```
<ToggleButton android:id="@+id/toggle1"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

- One thing that has been consistent throughout this example is that each view has the id attribute set to a particular value, such as in the case of the Button:

```
<Button android:id="@+id/btnSave"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/save" />
```

Different Views used

- The id attribute is an identifier for a view, which allows it to be retrieved using the View.findViewById() or Activity.findViewById() methods.

Program: **ViewsProgram**

Handling View Events

- To handle the events fired by each view, first must programmatically locate the view that created during the onCreate() event.
- This can be done using the findViewById() method (belonging to the Activity base class).
- By supplying the ID of the view to findViewById() method.

```
//---Button view---
Button btnOpen = (Button) findViewById(R.id.btnOpen);
```

- The setOnClickListener() method registers a callback to be invoked later when the view is clicked:

```
btnOpen.setOnClickListener(new View.OnClickListener() {
   public void onClick(View v) {
   displayToast("You have clicked the Open button");} );
```

- The onClick() method is called when the view is clicked.

Handling View Events

- To determine the state of the CheckBox, argument must be typecast in the onClick() method to a CheckBox and then verify its isChecked() method to see if it is checked:

```
CheckBox checkBox = (CheckBox) findViewById(R.id.chkAutosave);
checkBox.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        if (((CheckBox)v).isChecked())
            DisplayToast("CheckBox is checked");
        else
            DisplayToast("CheckBox is unchecked"); }
        });
```

Handling View Events

- For the RadioButton, setOnCheckedChangeListener() method is used on the RadioGroup to register a callback to be invoked when the checked RadioButton changes in this group.
- The following code will work for two radio buttons, as in a yes/no selection on a form:

```
//---RadioButton---
 RadioGroup radioGroup = (RadioGroup) findViewById(R.id.rdbGp1);
   radioGroup.setOnCheckedChangeListener(
       new OnCheckedChangeListener() {
   public void onCheckedChanged(RadioGroup group, int checkedId) {
    RadioButton rb1 = (RadioButton) findViewById(R.id.rdb1);
       if (rb1.isChecked()) {
              DisplayToast("Option 1 checked!");
       } else {
              DisplayToast("Option 2 checked!");
       } } } );
```

Handling View Events

- When a RadioButton is selected, the onCheckedChanged() method is fired. Within it, locate individual RadioButton views and then call each isChecked() method to determine which RadioButton is selected.
- Alternatively, the onCheckedChanged() method contains a second argument that contains a unique identifier of the selected RadioButton.
- The ToggleButton works just like the CheckBox.

Handling View Events

- There is another way to handle view events.
- Using the Button as an example, one can add an attribute called onClick:

```
<Button android:id="@+id/btnSave"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/save"
android:onClick="btnSaved_clicked"/>
```

- The onClick attribute specifies the click event of the button.
- The value of this attribute is the name of the event handler.
- Therefore, to handle the button’s click event, you simply create a method called btnSaved_clicked, as shown in the following example (note that the method must have a single parameter of type View):

Handling View Events

```
public class BasicViews1Activity extends Activity {
       public void btnSaved_clicked (View view) {
       DisplayToast("You have clicked the Save button1");
       }
       /** Called when the activity is first created. */
       @Override
       public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
       }
       private void DisplayToast(String msg) {
              Toast.makeText(getBaseContext(), msg,
              Toast.LENGTH_SHORT).show();
       }
}
```

ProgressBar View

- The ProgressBar view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background.
- For example, you might be downloading some data from the web and need to update the user about the status of the download.
- In this case, the ProgressBar view is a good choice.

Program: **ProgressBarView**

- The default mode of the ProgressBar view is indeterminate —that is, it shows a cyclic animation.
- This mode is useful for tasks that do not have specific completion times, such as when you are sending some data to a web service and waiting for the server to respond.
- If you simply put the <ProgressBar> Element in your main.xml file, it continuously displays a spinning icon.
- It is your responsibility to stop it when your background task has completed.

ProgressBar View

- The code added to the MainActivity.java file shows how you can spin off a background thread to simulate performing some long-running tasks. To do so, use the Thread class with a Runnable object.
- The run() method starts the execution of the thread, which in this case calls the doSomeWork() method to simulate doing some work. When the simulated work is done (after about five seconds), use a Handler object to send a message to the thread to dismiss the ProgressBar:

Program: CustomProgressBar

AutoCompleteTextView View

- The AutoCompleteTextView is a view that is similar to EditText (in fact it is a subclass of EditText), except that it automatically shows a list of completion suggestions while the user is typing.

Program: AutoCompleteTextView

- Create a String array containing a list of flowers in MainActivity.java class.
- The ArrayAdapter object manages the array of strings that are displayed by the AutoCompleteTextView.
- In this example, set the AutoCompleteTextView to display in the simple_dropdown_item_1line mode.

AutoCompleteTextView View

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>
(this,android.R.layout.simple_dropdown_item_1line, flowers);
```

- The setThreshold() method sets the minimum number of characters the user must type before the suggestions appear as a drop-down menu:

```
textView.setThreshold(3);
```

- The list of suggestions to display for the AutoCompleteTextView is obtained from the ArrayAdapter object:

```
textView.setAdapter(adapter);
```

Using Picker Views

- Selecting a date and time is one of the common tasks you need to perform in a mobile application.
- Android supports this functionality through the TimePicker and DatePicker views.

TimePicker View:

- The TimePicker view enables users to select a time of the day, in either 24-hour mode or AM/PM mode.
- When you are creating the project for this sample, be sure that you choose an SDK that is level 23 or greater.

Using Picker Views

Program: TimePicker

- The TimePicker displays a standard UI to enable users to set a time.
- By default, it displays the time in the AM/PM format.
- If you want to display the time in the 24-hour format, you can use the setIs24HourView() method.
- To programmatically get the time set by the user, use the getHour() and getMinute() methods.
- It’s better to display it in a dialog window because after the time is set, the window disappears and doesn’t take up any space in an activity.

Program: TimePickerDialog

TimePicker Dialog

- To display a dialog window, you use the showTimeDialog() method:

```
showTimeDialog();
```

- When the showTimeDialog() method is called, it creates a new instance of the TimePickerDialog class, passing it the current context, the callback, the initial hour and minute.
- It also determines whether the TimePicker should be displayed in 24-hour format.

```
public void showTimeDialog(){
new TimePickerDialog(MainActivity.this,dialogListener,
cal.get(Calendar.HOUR_
OF_DAY), cal.get(Calendar.MINUTE), false).show();
{ }; }
```

DatePicker View

- Another view that is similar to the TimePicker is the DatePicker.
- Using the DatePicker, user can enable users to select a particular date on the activity.

Program: DatePicker

- Like the TimePicker, you can also display the DatePicker in a dialog window.

Program: DatePickerDialog

Using List Views to Display Long Lists

- List views are views that enable to display a long list of items.
- In Android, there are two types of list views: ListView and SpinnerView.
- Both are useful for displaying long lists of items.

ListView View

- The ListView displays a list of items in a vertically scrolling list.

Program: LongListView

The first thing to notice in this example is that the MainActivity class extends the ListActivity class.

ListView View

- The ListActivity class extends the Activity class and displays a list of items by binding to a data source.
- Also note there is no need to modify the activity_main.xml file to include the ListView, because the ListActivity class itself contains a ListView
- With the onCreate() method, you don't need to call the setContentView() method to load the UI from the main.xml file:

```
//setContentView(R.layout.activity_main);
```

- In the onCreate() method, you use the setListAdapter() method to programmatically fill the entire screen of the activity with a ListView.
- The ArrayAdapter object manages the array of strings that are displayed by the ListView.

ListView View

```
setListAdapter(new ArrayAdapter<String>(this,
     android.R.layout.simple_list_item_1, flowers));
```

- The onListItemClick() method is fired whenever an item in the ListView has been clicked:

```
public void onListItemClick( ListView parent, View v,
int position, long id){
Toast.makeText(this,"You have selected " +
flowers[position], Toast.LENGTH_SHORT).show();
}
```

Customizing the ListView

- The ListView is a versatile view that can be customized further.

Program: CustomListView

- To programmatically get a reference to the ListView object, use the getListView() method, which fetches the ListActivity's list view.
- This is necessary so that you can programmatically modify the behavior of the ListView.
- In this case, the setChoiceMode() method is used to specify how the ListView should handle a user's click. For this example, set the setChoiceMode() to ListView.CHOICE_MODE_MULTIPLE, which allows the user to select multiple items:

```
ListView lstView = getListView();
lstView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
```

Customizing the ListView

- A very cool feature of the ListView is its support for filtering.
- When filtering is enabled through the setTextFilterEnabled() method, users can type on the keypad and the ListView automatically filters the items to match what was typed:

```
lstView.setTextFilterEnabled(true);
```

Storing Items in the strings.xml File

- Although the previous example shows that the list of flower names is stored in an array.
- In a real-life application it is recommended to either retrieve them from a database or at least store them in the strings.xml file.

Program: ItemStringXML

- With the names now stored in the strings.xml file, you can retrieve it programmatically in the MainActivity.java file using the getResources() method:

```
flowers =  getResources().getStringArray(R.array.flowers_array);
```

- In general, you can programmatically retrieve resources bundled with your application using the getResources() method.

Make items in a ListView selectable

- This example demonstrated how to make items in a ListView selectable.

Program: ItemSelectable

Using the Spinner View

- The ListView displays a long list of items in an activity, but you might want the user interface to display other views, meaning you do not have the additional space for a full-screen view, such as the ListView.
- In such cases, the SpinnerView can be used.
- The SpinnerView displays one item at a time from a list and enables users to choose from them.

Using the Spinner View

- The onNothingSelected() method is fired when the user presses the back button, which dismisses the list of items displayed.
- In this case, nothing is selected so you do not need to do anything.
- Instead of displaying the items in the ArrayAdapter as a simple list, you can display them using radio buttons.
- To do so, modify the second parameter in the constructor of the ArrayAdapter class:

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_single_choice, flowers);
```

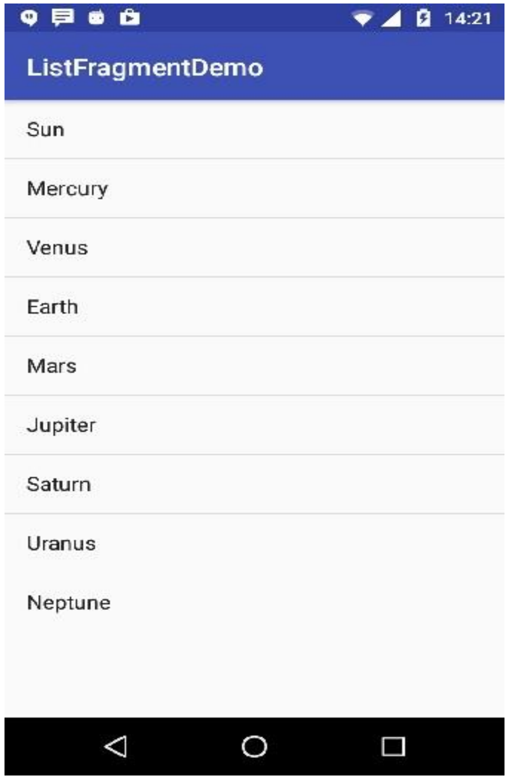
Program: SpinnerView (2 Versions- with / without radio button)

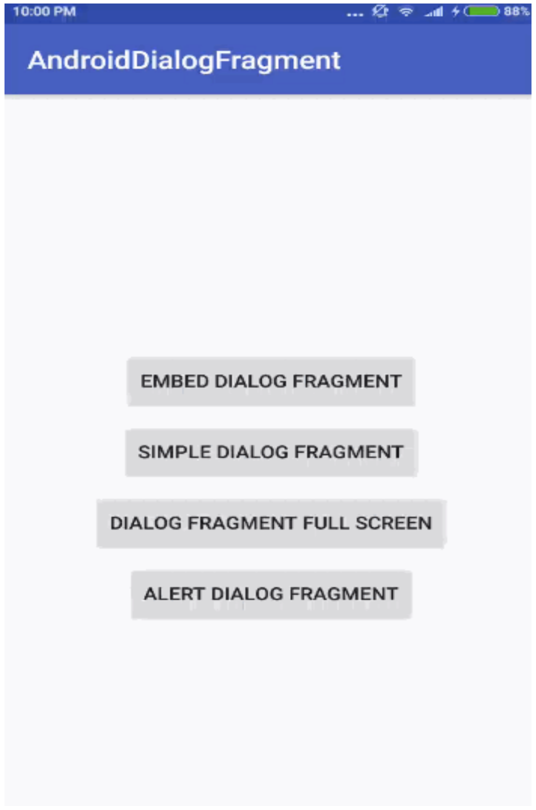
Understanding Specialized Fragments

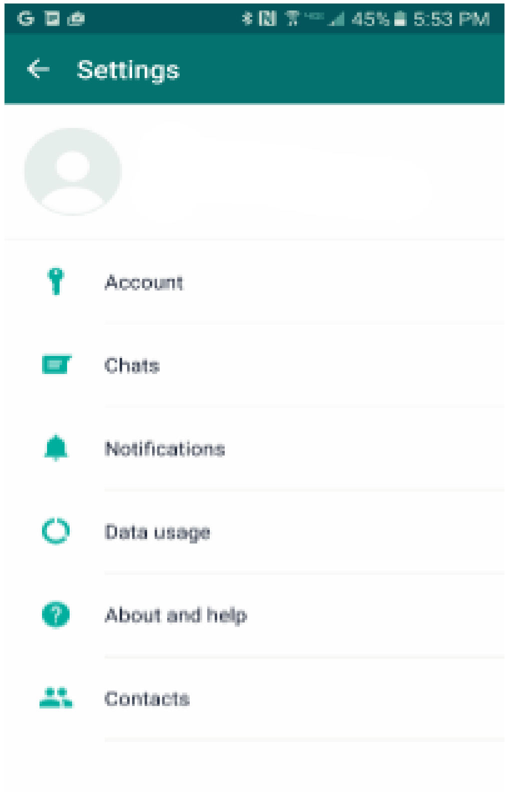
- Fragments allows to customize the user interface of Android application by dynamically rearranging fragments to fit within an activity.
- This enables to build applications that run on devices with different screen sizes.
- Fragments are really “mini-activities” that have their own life cycles.
- To create a fragment, a class needed that extends the Fragment base class.
- In addition to the Fragment base class, you can also extend from some other subclasses of the Fragment base class to create more specialized fragments.

Understanding Specialized Fragments

- The three subclasses of Fragment are:
  - ListFragment
  - DialogFragment
  - PreferenceFragment







Using a ListFragment

- A list fragment is a fragment that contains a ListView, which displays a list of items from a data source, such as an array or a Cursor.
- A list fragment is useful because it's common to have one fragment that contains a list of items (such as a list of RSS postings), and another fragment that displays details about the selected posting.
- To create a list fragment, you need to extend the ListFragment base class.

Program: ListFragment

- First, create the XML file for the fragment by adding a ListView element.

Using a ListFragment

- To create a list fragment, the Java class for the fragment must extend the ListFragment base class:

```
public class Fragment1 extends ListFragment {
}
```

- Then declare an array to contain the list of flower names in the activity
- In the onCreate() event, use the setListAdapter() method to programmatically fill the ListView with the content of the array.
- The ArrayAdapter object manages the array of strings that are displayed by the ListView.
- In this example, set the ListView to display in the simple_list_item_1 mode

Using a ListFragment

```
@Override
 public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setListAdapter(new ArrayAdapter<String>(getActivity(),
   android.R.layout.simple_list_item_1, flowers));
}
```

- The onListItemClick() method is fired whenever an item in the ListView is clicked:

```
public void onListItemClick(ListView parent, View v,
int position, long id)
{
Toast.makeText(getActivity(),
"You have selected " + flowers[position],
Toast.LENGTH_SHORT).show();
}
```

- Finally, add two fragments to the activity. Note the height of each fragment:

Using a DialogFragment

- A dialog fragment floats on top of an activity and is displayed modally.
- Dialog fragments are useful when you need to obtain the user's response before continuing with execution.
- To create a dialog fragment, you must extend the DialogFragment base class.

Program: DialogFragment

- To create a dialog fragment, Java class first must extend the DialogFragment base class:

```
public class Fragment1 extends DialogFragment {
}
```

Using a DialogFragment

- Create an alert dialog, which is a dialog window that displays a message with optional buttons.
- Within the Fragment1 class, define the newInstance() method:

```
static Fragment1 newInstance(String title) {
Fragment1 fragment = new Fragment1();
Bundle args = new Bundle();
args.putString("title", title);
fragment.setArguments(args);
return fragment;
}
```

- The newInstance() method allows a new instance of the fragment to be created.
- Also, this method accepts an argument specifying the string (title) to display in the alert dialog.
- The title is then stored in a Bundle object for use later.

Using a DialogFragment

- Define the onCreateDialog() method, which is called after onCreate() and before onCreateView():

```
public Dialog onCreateDialog(Bundle savedInstanceState) {
String title = getArguments().getString("title");
return new AlertDialog.Builder(getActivity())
.setIcon(R.mipmap.ic_launcher)
.setTitle(title)
.setPositiveButton("OK", new DialogInterface.OnClickListener()
{ public void onClick(DialogInterface dialog,
int whichButton) { ((MainActivity)
getActivity()).doPositiveClick(); } })
.setNegativeButton("Cancel",new
DialogInterface.OnClickListener() {
public void onClick(DialogInterface dialog,
int whichButton) { ((MainActivity)
getActivity()).doNegativeClick(); } }).create();
}
```

Using a DialogFragment

- Create an alert dialog with two buttons: OK and Cancel. The string to be displayed in the alert dialog is obtained from the title argument saved in the Bundle object.
- To display the dialog fragment, you create an instance of it and then call its show() method:

```
Fragment1 dialogFragment = Fragment1.newInstance(
"Are you sure you want to do this?");
dialogFragment.show(getFragmentManager(), "dialog");
```

- Implement two methods, doPositiveClick() and doNegativeClick(), to handle the user clicking the OK or Cancel buttons, respectively:

Using a DialogFragment

```
public void doPositiveClick() {
//---perform steps when user clicks on OK---
Log.d("DialogFragmentExample", "User clicks on OK");
}
public void doNegativeClick() {
//---perform steps when user clicks on Cancel---
Log.d("DialogFragmentExample", "User clicks on Cancel");
}
```

Using a PreferenceFragment

- Typically, in Android applications you provide preferences for users to personalize the application.
- For example, you might allow users to save the login credentials that they use to access their web resources.
- Also, you could save information, such as how often the feeds must be refreshed (for example, in an RSS reader application), and so on.
- In Android, you can use the PreferenceActivity base class to display an activity for the user to edit the preferences.
- In Android 3.0 and later, you can use the PreferenceFragment class to do the same thing.

Program: Preference Fragment

Using a PreferenceFragment

- To create a list of preferences in your Android application, first need to create the preferences.xml file and populate it with the various XML elements.
- Before that, if you are using latest android studio and API above 28, Add

```
dependencies {  ...
implementation 'androidx.appcompat:appcompat:1.1.0'
implementation "androidx.preference:preference:1.1.0"
... }
```

to build.gradle (Module:app)

- This XML file defines the various items that you want to persist in your application.

Using a PreferenceFragment

- To create the preference fragment, you must extend the PreferenceFragment base class:

```
public class Fragment1 extends PreferenceFragment {
}
```

- To load the preferences file in the preference fragment, use the addPreferencesFromResource() method:

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
//---load the preferences from an XML file---
addPreferencesFromResource(R.xml.preferences);
}
```

Using a PreferenceFragment

- To display the preference fragment in your activity, you can make use of the FragmentManager and the FragmentTransaction classes:

```
FragmentManager fragmentManager = getFragmentManager();
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
Fragment1 fragment1 = new Fragment1();
fragmentTransaction.replace(android.R.id.content,
fragment1);
fragmentTransaction.addToBackStack(null);
fragmentTransaction.commit();
```

- You need to add the preference fragment to the back stack using the addToBackStack() method so that the user can dismiss the fragment by clicking the Back button.