<u>UNIT – 2</u>

Working with the User Interface using views:

Understanding the Components of a Screen-Adapting to Display Orientation-Managing Changes to Screen Orientation- Utilizing the Action Bar-Creating the User Interface Programmatically Listening for UI Notification.

Understanding the Components of a Screen:

- The basic unit of an Android application is an *activity*, which displays the UI of your application.
- you define your UI using an XML file (for example, the activity_main.xml file located in the res/layout folder of your project)
- During runtime, you load the XML UI in the onCreate() method handler in your Activity class, using the setContentView() method of the Activity class:

@Override

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView (R.layout.main);
```

```
}
```

Views and View groups:

- An activity contains *views* and *ViewGroups*. A View usually draws something the user can see and interact with. A view is a widget that has an appearance on screen. Examples of views are buttons, labels, and text boxes.
- A view derives from the base class **android.view.View.**
- In ViewGroup, one or more Views can be grouped together. A ViewGroup provides the layout, in which you can set the order of the appearance and sequence of the Views. Some examples of ViewGroups are LinearLayout and FrameLayout.



> It is derived from the base class **android.view.ViewGroup**.

Following are common attributes and will be applied to all the layouts:

Sr.No	Attribute & Description		
1	android:id This is the ID which uniquely identifies the view.		
2	android:layout_width This is the width of the layout.		
3	android:layout_height This is the height of the layout		
4	android:layout_marginTop This is the extra space on the top side of the layout.		
5	android:layout_marginBottom This is the extra space on the bottom side of the layout.		
6	android:layout_marginLeft This is the extra space on the left side of the layout.		
7	android:layout_marginRight This is the extra space on the right side of the layout.		

8	android:layout_gravity This specifies how child Views are positioned from top bottom left and write from the window
9	android:layout_weight This specifies how much of the extra space in the layout should be allocated to the View.It distributes the available space among the child views
10	android:layout_x This specifies the x-coordinate of the layout.
11	android:layout_y This specifies the y-coordinate of the layout.
12	android:layout_width This is the width of the layout.
13	android:paddingLeft This is the left padding filled for the layout.
14	android:paddingRight This is the right padding filled for the layout.
15	android:paddingTop This is the top padding filled for the layout.
16	android:paddingBottom This is the bottom padding filled for the layout.

Android supports the following ViewGroups.

- Layout is used to arrange view and group of views visually on the screen.
- 1. ConstriantLayout
- 2. LinearLayout

- 3. AbsoluteLayout
- 4. TableLayout
- 5. RelativeLayout
- 6. FrameLayout
- 7. Scroll View
- 8. GridView
- 9. ListView

ConstriantLayout:

- ConstraintLayout is a ViewGroup subclass; whenever you open the android studio framework the application will be present in the constraint layout. In this we have to set the constraint in all four sides.
- This is the default layout. It's typically used to manage the positioning of individual views rather than entire groups of views.
- Ex: Designing a login screen for a mobile app using constraint layout in Android.



For example, you can specify that the logo is centered horizontally in the parent, the username input field is centered below the logo, the password input field is centered below the username field, and so on.

This way, no matter what device the app is running on, whether it's a small phone or a large tablet, the login screen will adjust dynamically to fit the screen size while maintaining the specified layout constraints.

LinearLayout:

The LinearLayout arranges views in a single column or a single row. Child views can be arranged either horizontally or vertically in a single direction, which explains the need for two different layouts—one for horizontal rows of views and one for vertical columns of views. You can specify the layout direction with the <u>android:orientation</u> attribute.



- The root layout is a LinearLayout with android:orientation="horizontal", which means child views, will be placed side by side. Suitable for toolbars, menus, and other horizontally aligned components.
- The root layout is a Linear Layout with android:orientation="vertical", which means child views will be placed one below the other. Suitable for forms, lists, and other vertically aligned components.





AbsoluteLayout:

- An Absolute layout allows you to specify the exact location. i.e., X and Y coordinate of its children with respect to the origin at the top left corner of the layout.
- > android:layout_x
 - This specifies the x-coordinate of the view.
- android:layout_y
 - This specifies the y-coordinate of the view.

```
android:layout_x="50px"
android:layout_y="361px"
```

Absolute	eLayout Example
AbsoluteLay	outExample
User Name	
Password	
	LOG IN



TableLayout :

- TableLayout is a view that groups views into rows and columns. You use the <TableRow> element to designate a row in the table. Each row can contain one or more views.
- In android, TableLayout will position its children elements into rows and columns and it won't display any border lines for rows, columns or cell.





RelativeLayout:

- RelativeLayout is a view group that displays child views in relative positions. The position of each view can be specified as relative to sibling elements or relative to the parent.
- Each view embedded within the RelativeLayout has attributes that enable it to align with another view.
- ➤ These attributes are as follows:
 - **layout_alignParentTop:** Aligns the top edge of the view with the top edge of the parent.
 - **layout_alignParentStart:** Aligns the start edge of the view with the start edge of the parent.
 - **layout_alignStart:** Aligns the start edge of the view with the start edge of another specified view.
 - **layout_alignEnd:** Aligns the end edge of the view with the end edge of another specified view.
 - **layout_below:** Positions the view directly below another specified view.

• **layout_centerHorizontal:** Centers the view horizontally within the parent.



FrameLayout:

- The FrameLayout is a placeholder on screen that you can use to display a single view. Views that you add to a FrameLayout are always anchored to the top left of the layout.
- FrameLayout is a ViewGroup subclass, The FrameLayout is the most basic of the Android layouts. FrameLayouts are built to hold one view.
- You can add multiple views to a FrameLayout, but each is stacked on top of the previous one. This is when you want to animate a series of images, with only one visible at a time.
- You can, however, add multiple children to a FrameLayout and control their position within the FrameLayout by assigning gravity to each child, using the android:layout_gravity attribute. android:gravity="center".



Example: Image Carousel with FrameLayout Imagine you want to create an image carousel where multiple images are displayed one after another with a fade-in animation, while a caption is shown on top of each image. FrameLayout is perfect for this because it allows stacking views and handling animations easily.

ScrollView

- A ScrollView is a special type of FrameLayout in that it enables users to scroll through a list of views that occupy more space than the physical display.
- The ScrollView can contain only one child view or ViewGroup, which normally is a LinearLayout. Android supports vertical scroll view as default scroll view. Vertical scrollview scrolls element s vertically. Android uses Horizontal ScrollView for scrolls element horizontally.

GridView

GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid.

ListView

- ListView is a view group that displays a list of scrollable items.
 wrap_content: Tells the view to size itself to the dimensions required by its content.
- Example: If you have a TextView with android:layout_width="wrap_content", the TextView will expand just enough to fit the text it contains.

match_parent (previously fill_parent):

- Definition: Tells the view to become as big as its parent view, filling the entire available space.
- Example: If you have a Button with android:layout_width="match_parent", the Button will stretch to fill the width of its parent layout.

These attributes are important for controlling the layout and appearance of views within your Android app's user interface.

attribute	Description		
layout_width	Specifies the width of the view or ViewGroup		
layout_height	Specifies the height of the view or ViewGroup		
lavout marginTon	Specifies extra space on the top side of the view		
	or ViewGroup		
lavout marginBottom	Specifies extra space on the bottom side of the		
	view or ViewGroup		
lavout margin oft	Specifies extra space on the left side of the view		
	or ViewGroup		
lavout marginRight	Specifies extra space on the right side of the		
	view or ViewGroup		
layout_gravity	Specifies how child views are positioned		
lavout woight	Specifies how much of the extra space in the		
layout_weight	layout should be allocated to the view		
lavout v	Specifies the x-coordinate of the view or		
layout_x	ViewGroup		
lavout v	Specifies the y-coordinate of the view or		
layout_y	ViewGroup		

Common Attributes Used in Views and ViewGroups

Adapting to Display Orientation :

- As with almost all smartphones, Android supports two screen orientations: portrait and landscape.
- When the screen orientation of an Android device is changed, the current activity being displayed is destroyed and re-created automatically to redraw its content in the new orientation.
- In other words, the onCreate() method of the activity is fired whenever there is a change in screen orientation.
- Portrait mode is longer in height and smaller in width, whereas landscape mode is wider but smaller in height.
- Being wider, landscape mode has more empty space on the right side of the screen.
- At the same time, some of the controls don't appear because of the smaller height.
- Thus, controls need to be laid out differently in the two screen orientations because of the difference in the height and width of the two orientations.

There are two ways to handle changes in screen orientation: Anchoring Views/Controls—

- 1. Anchoring involves setting the position of views (or controls) relative to the four edges of the screen. When the screen orientation changes, the controls/views do not disappear but are rearranged (the views adjust their positions accordingly, maintaining a consistent layout and avoiding disappearing off-screen) relative to the four edges.
- 2. For anchoring controls relative to the four edges of the screen, we use a RelativeLayout container. The controls are aligned relative to the edges of the container.

Defining layout for each mode—

• This approach involves creating separate XML layout files for different screen orientations (portrait and landscape). Each layout file is tailored to suit the specific orientation, ensuring the UI is optimized for both modes.

3354demoAVD		
ScreenOrientationAppActivity		
Apple	S554demoAVD	
Mango	ScreenOrientationAppActivity	*2 🕯 8:22
Banana	Apple	-
Grapes	Mango	
Kiwi	Banana	

•This vertical arrangement makes a few of the Button controls disappear when the screen is in landscape mode. To use the blank space on the right side of the screen in landscape mode, we need to define another layout file.

S554:demoAVD		
³⁶ / 🖬 8:31		
ScreenOrientationAppActivity		
Apple	S554:demoAVD	
		³ 5⁄ 🙆 8:31
Mango	ScreenOrientationAppActivity	
Banana	Apple Mango	
Grapes	Banana Grapes Screen switched to Landcape mode	
Screen switched to Portrait mode	Kiwi	

Figure 3.18. (left) Controls in portrait mode, and (right) all controls are visible in landscape mode.

Resizing and Repositioning Views:

• This approach involves dynamically adjusting the size and position of views within a single layout file to respond to changes in screen orientation. Instead of switching layouts, you modify view properties programmatically.

ScreenOrien	[™] 2 € 6:58 tationAppActivity			
Apple	Mango			
Banana Kiwi		5554 demoAVD ScreenOrientationApp	oActivity	[™] 2 8 6:59
		Apple	Mango	
	Grapes	Banana		
		Kiwi		Grapes

Figure 3.16. (left) Controls in portrait mode, and (right) the controls in landscape mode

 Anchoring Views Anchoring can be easily achieved by using RelativeLayout. Consider the following main.xml file, which contains five Button views embedded within the element:

CODE:

<RelativeLayout android:layout_width="match_parent" android:layout_height="match_parent" xmlns:android="http://schemas.android.com/apk/res/android">

<!-- Button anchored to the top-left corner -->

<Button android:id="@+id/button1" android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="Top Left" android:layout_alignParentStart="true" android:layout_alignParentTop="true" />

<!-- Button anchored to the top-right corner -->

<Button

android:id="@+id/button2"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Top Right"
android:layout_alignParentTop="true"
android:layout_alignParentEnd="true" />

<!-- Button anchored to the bottom-left corner -->

<Button

android:id="@+id/button3"
android:layout_width="wrap_content"
android:layout_height="wrap_content"

```
android:text="Bottom Left"
android:layout_alignParentStart="true"
android:layout_alignParentBottom="true" />
```

<!-- Button anchored to the bottom-right corner -->

<Button android:id="@+id/button4" android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="Bottom Right" android:layout_alignParentEnd="true" android:layout_alignParentBottom="true" />

<!-- Button centered in the middle of the parent -->

<Button android:id="@+id/button5" android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="Middle" android:layout_centerVertical="true" android:layout_centerHorizontal="true" />

</RelativeLayout>

Button 1 (button1): Anchored to the top-left corner of the parent.Button 2 (button2): Anchored to the top-right corner of the parent.Button 3 (button3): Anchored to the bottom-left corner of the parent.Button 4 (button4): Anchored to the bottom-right corner of the parent.Button 5 (button5): Centered both vertically and horizontally in the parent.

Note the following attributes found in the various Button views:

➤ layout_alignParentStart—Aligns the view to the left of the parent view

- ➤ layout_alignParentEnd—Aligns the view to the right of the parent view
- ➤ layout_alignParentTop—Aligns the view to the top of the parent view
- ➤ layout_alignParentBottom—Aligns the view to the bottom of the parent view
- ➤ layout_centerVertical—Centers the view vertically within its parent view
- ➤ layout_centerHorizontal—Centers the view horizontally within its parent view

		⁵⊿ 🖬 9:34
My Application	n	
TOP LEFT		TOP RIGHT
	MIDDLE	
		2022011200
BOTTOM LEFT		BOTTOM RIGHT
\bigtriangledown	0	

FIGURE 4-18

0		21 🗄 9:35	
My Application			
TOP LEFT		TOP RIGHT	
		•	0
	MIDDLE		
			4
BOTTOM LEFT		BOTTOM RIGHT	

Managing Changes to Screen Orientation:

- What happens to an activity's state (data) when the device changes orientation Understanding Activity Behavior when Orientation Changes.
- Activity behavior Changes when the screen orientation Changes in Android, so when the device orientation changes, frist the activity will disappear for a millisecond when the onPause(),onStop(),onDestroy() methods are called.
- After a few milliseconds, the activity will be restarted when the onCreate(), onStart(),onResume() methods are called.

1. Using Android Studio, create a new Android project and name it Orientations.

2. Add the bolded statements in the following code to the activity_main.xml file:

```
<?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" >
<EditText
android:id="@+id/txtField1"
android:layout_width="fill_parent"
android:layout_height="wrap_content" />
<EditText
# android:id="@+id/txtField2"
android:layout_width="fill_parent"
android:layout_width="fill_parent"
android:layout_width="fill_parent"
android:layout_width="fill_parent"
```

3. Add the bolded statements in in the following code to the MainActivity.java file:

```
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
public class MainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("StateInfo", "onCreate");
    }
    @Override
    public void onStart() {
```

```
Log.d("StateInfo", "onStart");
super.onStart();
 }
(a)Override
public void onResume() {
Log.d("StateInfo", "onResume");
super.onResume();
 }
(a)Override
public void onPause() {
Log.d("StateInfo", "onPause");
super.onPause();
 }
(a)Override
public void onStop() {
Log.d("StateInfo", "onStop");
super.onStop();
 }
(a)Override
public void onDestroy() {
Log.d("StateInfo", "onDestroy");
super.onDestroy();
 }
(a)Override
public void onRestart() {
Log.d("StateInfo", "onRestart");
super.onRestart();
 }
}
   \blacktriangleright Press F11 to debug
```

- Enter some text into the two EditText views
- Change the orientation of the Android Emulator by pressing Ctrl+F11.
- Note that the text in the first EditText view is still visible, while the second EditText view is now empty.

How It Works

• From the output shown in the logcat console, it is apparent that when the device changes orientation.

The activity is destroyed:

- 12-15 12:39:37.846: D/StateInfo(557): onPause
- 12-15 12:39:37.846: D/StateInfo(557): onStop
- 12-15 12:39:37.866: D/StateInfo(557): onDestroy

It is then re-created:

- 12-15 12:39:38.206: D/StateInfo(557): onCreate
- 12-15 12:39:38.216: D/StateInfo(557): onStart
- 12-15 12:39:38.257: D/StateInfo(557): onResume
 - For example, the user might change orientation while entering some text into an EditText view. When this happens, any text inside the EditText view is persisted and restored automatically when the activity is re-created. Conversely, if you do not name the EditText view using the **android:id** attribute, the activity isn't able to persist the text currently contained within it.





<u>Utilizing the Action Bar:</u>

- Android ActionBar is a menu bar that runs across the top of the activity screen in android. Android ActionBar can contain menu items which become visible when the user clicks the "menu" button.
- Action bar is a combination of on-screen action items and overflow options. Toolbar/Action Bar: This is a bar at the top of the screen in Android apps, which contains the application logo, title, and navigation/menu items. It often hosts contextual actions or options relevant to the current screen.

In general an ActionBar consists of the following four components:

- App Icon: App branding logo or icon will be displayed here.
- **View Control**: A dedicated space to display Application title. Also provides option to switch between views by adding spinner or tabbed navigation.
- Action Buttons: Some important actions of the app can be added here
- Action Overflow: All unimportant action will be shown as a menu.





Spinner: A spinner typically appears as a small rectangular box with an arrow icon at the right side. When the user taps on the spinner, a dropdown list of items appears below it, allowing the user to select one.

Dropdown Menu: A dropdown menu, on the other hand, is a pop-up menu that appears below the anchor view (such as a button or text field) when the user interacts with it. It doesn't have a visible box or arrow icon associated with it.

How to set and Changing the Action Bar title:

Setting Toolbar as An ActionBar

Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar); // get the reference

of Toolbar

setSupportActionBar(toolbar); // Setting/replace toolbar as the ActionBar

This example demonstrates How to set title for action bar in

android.

```
if (getSupportActionBar() != null) {
```

getSupportActionBar().setTitle("Home");

getSupportActionBar().setSubtitle("sairam");

getSupportActionBar().setDisplayShowHomeEnabled(true);

getSupportActionBar().setIcon(R.drawable.ic_launcher); // Replace with

your icon

```
}
textView.setText("Title is Home");
```

} **Changing the Action Bar title:** if (actionBar != null) { actionBar.setTitle(newTitle); }

Add items to the Action bar:

<item android:id="@+id/miProfile" android:icon="@drawable/ic_profile" app:showAsAction="ifRoom | withText" android:title="Profile"> </item>

11:07 ad ad ≋ 1 KB/s ∰ Home sairam	5) ® 📧	11:07 .al al ஆ 1 KB/s ‱ Home	a®®
tittle is Home		tittle is Home	

<u>Creating the user interface programmatically:</u>

- 1. In Android you can create the user interface programmatically by instantiating and configuring UI components in your java or kotlin code rather than using XML layout files.
- 2. So far, we have seen all the UIs which have created, using XML file but we can create the user interface programmatically. This is useful; When UI needs to be dynamically generated during runtime.

Ex: If you are creating an app for the air ticket reservation system and your app is supposed to display the seats for each way's travel, using the Buttons. In this case, you will have to dynamically generate the UI code, which is based on the air travel selected by the user.

To create a user interface programmatically in Android Studio, you typically follow these steps:

1. **Create a Layout Container:** You'll need a layout container to hold your UI Elements programmatically. This can be a LinearLayout, RelativeLayout, ConstraintLayout, etc. You can create it directly in your activity's XML layout file or Programmatically in your Java/Kotlin code.

2. **Instantiate UI Elements:** Create instances of the UI elements you want to Include in your layout. For example, TextView, Button, EditText, etc.

3. **Set Layout Parameters:** For each UI element, create layout parameters Specifying how it should be positioned and sized within the layout container.

4. Add UI Elements to Layout: Add the UI elements to the layout container Using the addView() method.

5. **Set Content View:** If you created the layout container programmatically, set it as the content view of your activity using the setContentView () method.

Creating the UI via Code:

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.support.v7.widget.LinearLayoutCompat;

import android.widget.Button;

import android.widget.LinearLayout;

import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
 @Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

LinearLayoutCompat.LayoutParams params =

newLinearLayoutCompat.LayoutParams (

LinearLayoutCompat.LayoutParams.WRAP_CONTENT,

LinearLayoutCompat.LayoutParams.WRAP_CONTENT);

LinearLayout layout = new LinearLayout(this); layout.setOrientation(LinearLayout.VERTICAL); TextView tv = new TextView(this); tv.setText("This is a TextView"); tv.setLayoutParams(params); Button btn = new Button(this); btn.setText("This is a Button"); btn.setLayoutParams(params);

layout.addView(tv); layout.addView(btn); LinearLayoutCompat.LayoutParams layoutParam = new LinearLayoutCompat.LayoutParams(LinearLayoutCompat.LayoutParams.WRAP_CONTENT, LinearLayoutCompat.LayoutParams.WRAP_CONTENT); this.addContentView(layout, layoutParam);



Listening for UI notifications:

- ➤ Users interact with your UI at two levels:
 - Activity level
 - View level

Activity level :

- At the activity level, the Activity class exposes methods that you can override. Some common methods that you can override in your activities include the following:
 - **On Key Up():** This is called when a key was released. This is not handled by any of the views inside the activity.
 - **On Key Down():** This is called when a key was pressed. This is not handled by any of the views inside the activity.
 - **On Menu Item Selected():** This is called when any item of the menu panel is pressed by user.
 - **On Menu Opened():** This method is called when user opens the panel's menu.

View Level:

When any user interacts with a view, the corresponding view fires event. When a user touches a button or an image button or any such view we have to service the related service so that appropriate action can be performed. For this, events need to be registered.