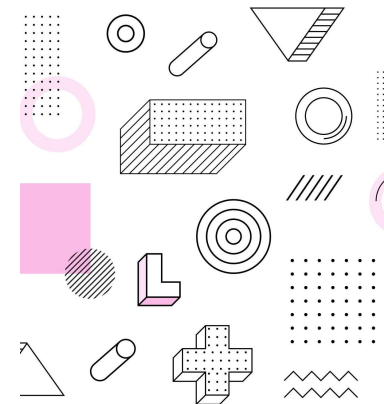
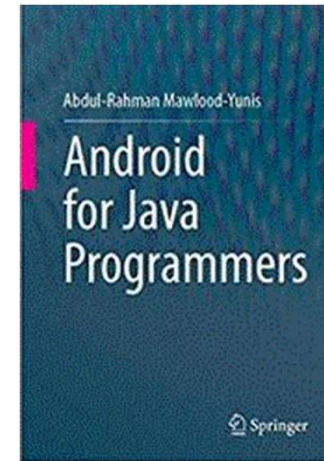
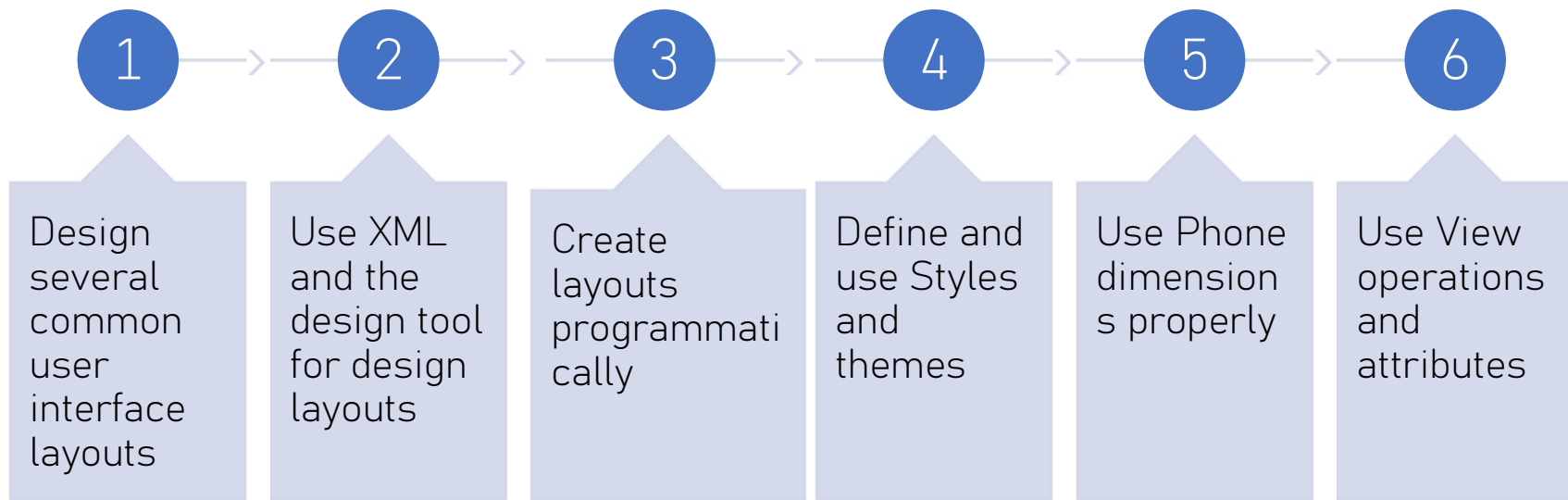


# Chapter 06: User Interface essential classes, Layouts, Styles, Themes, and Dimensions

---



# What You Will Learn in This Chapter



# Check out the demo project

- How to run the code:  
unzip the code in a folder of your choice, then in Android Studio click File->import->Existing Android code into the workspace

# 6.1 Introduction

---



From your personal experience using mobile apps, you may have noticed that the user interface is an important part of the app and is vital in its success



UI is a large topic that would be difficult to cover in-depth



We will study Views, Layouts, Widgets, and other components that enable the creation of nice-looking and user-friendly interfaces for Android apps

## Part 1: Essential UI Classes and Properties

---

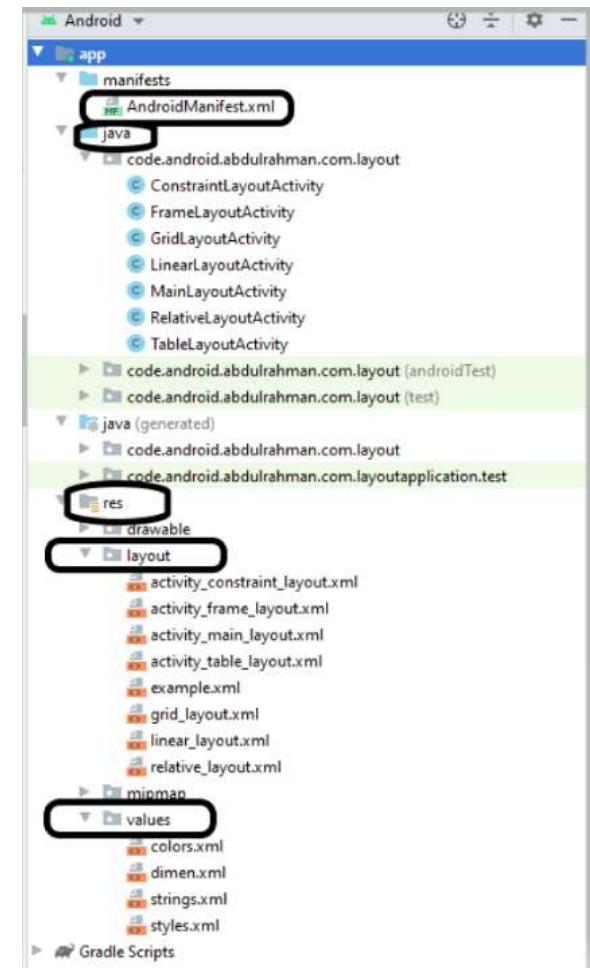
We start by trying to understand the directory structure of an app project in Android studio and user interface classes that are widely used in all apps, including our demo app

## 6.2 Android Project Structure

The **Java folder** holds the MainActivity and another six activities, one for each button on the demo app's main screen

The resource files are stored in the '**res**' folder and they are strings, dimensions, images, menu text, colours, and style files

The project directory structure also includes the **manifest XML file** which wires all components and resources together to make the app work



## 6.3 Views

---

01

In Java, we use the **object** to refer to almost everything

02

When developing Android apps, you are dealing with UI components and almost everything is a visual object,

03

you can use **View** instead of using the object to refer to any UI component of an app

## 6.3 Views

---

**View** is an Android class that is an essential building block of Android's user interface

It allows you to create a layout or display area of your app to put View subclasses and UI components inside it

It occupies the rectangular area of your device

It is in charge of drawing and handling user interactions



## 6.3.1 Views Listeners

---



Oftentimes when you include a View in your app, it is used to interact with the user



You as a programmer will set up listeners that will be called when an event gets fired



For example, when a button is pressed by the user, the input data can be saved



For dealing with event firing and handling, Android follows the exact Java listener handling approach

## 6.3. 1 Views Listeners

---

First you register the component that will fire the event

Second, you write the method which will get executed once the event is fired

This method execution is called event handling

## 6.3 View Properties

---

---

Each View has multiple attributes or properties. Some of these properties need to be set when defining View objects inside the layout.

---

These properties can be set in the XML layout file or programmatically in the code.

---

For example, the View class has attributes such as id, width, height, background color, etc

---

Defining Views and their properties in the XML layout file keeps the code clean and compact and is the most common practice.

## 6.3.2 View Properties

---

**Examples of properties that you can set when defining a View object include setting**

1. text
2. positions
3. margins
4. padding
5. dimensions
6. background colours
7. text font size
8. style
9. etc.



## 6.4 View Class Examples

**EditText** is a user interface class for entering text.

The EditText definition is listed below.

<EditText

android:id="@+id/editName"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

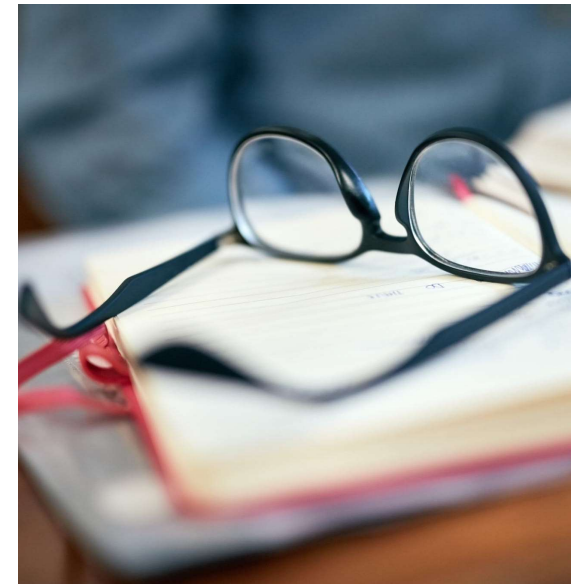
android:layout\_margin="5dp"

android:hint="@string/name\_hint"

android:inputType="textCapWords"

android:singleLine="true" >

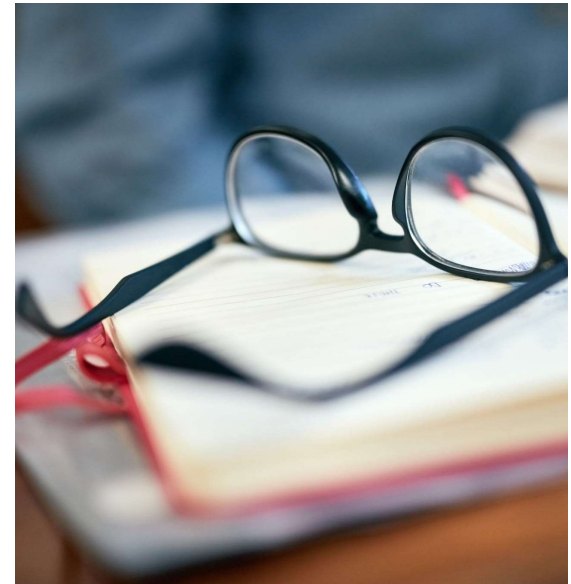
</EditText>



## 6.4 View Class Examples

### View attributes

id	"@+id/editName"
width	"match_parent"
height	"wrap_content"
margin	"5dp"
hint	"@string/name_hint"
inputType	"textCapWords"
singleLine	"true"



## 6.4 android:id = "@+id/editName"

---

To define an **id** attribute, i.e., assign a View or a resource an id, you need to include the at-symbol "@" and "+" signs in the attribute declaration

The plus-symbol (+) means a new resource name must be created and added to the R.java file. That is, editName does not exist and needs to be created as a static final int field of the R.java class

The part of id definitions that you refer to in your code is the name part only. E.g., `EditText editText = findViewById(R.id.editName);`

When referencing a view id there is no need to use the plus symbol the "@" and the Android namespace are needed e.g.,  
`android:layout_below="@id/editName"`

## 6.4 View attributes

---

**`android:layout_width="match_parent"`.**

The `match_parent` attribute value means that the view will be as wide as its parent minus any padding

**`android:layout_height="wrap_content"`.**

This means that the view is big enough (width and high) to enclose its content plus any padding

**`android:layout_margin="5dp"`.**

This is the space outside of the border of the view (e.g., a button) and between what is next to, or around, the view



## 6.4 View attributes

---

**android:hint="@string/name\_hint**. This is a hint of what the user should enter into EditText.

For example, you can use `android:hint=" enter your email here"`, on a form to hint the user that the field is for inserting an email address.

**android:inputType="textCapWords"**.

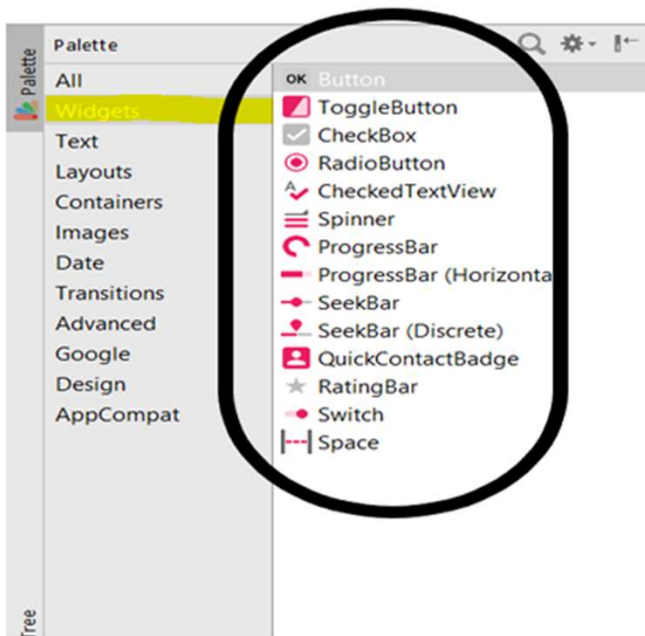
This is the user input where the letter of each word will begin with capital by default. For example, Bill Gates.

There are other types of input as well. These include **textEmailAddress** and **textAutoComplete**.

**android:singleLine="true"**. The input by the user is restricted to a single line.

**android:padding** Sets the padding, in pixels, of all four edges. Padding is defined as space between the edges of the view and the view's content.

## 6.5 Widget

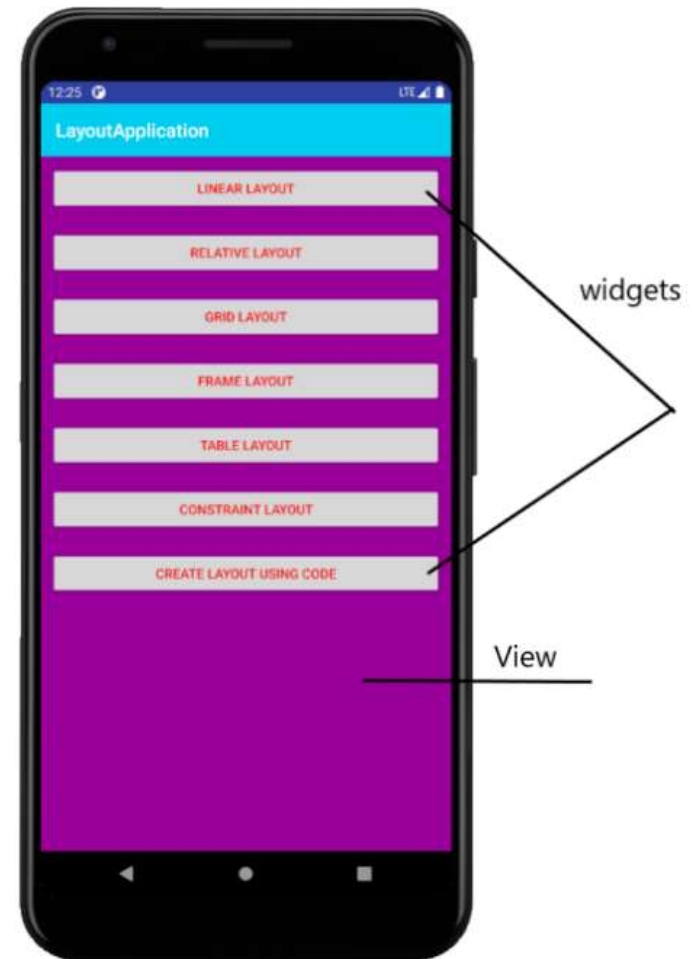


- several view subclasses are grouped in a package called **Widgets**.
- Widgets are fundamental control elements of the graphical user interface of an app. It is through these elements that the user can interact with the app.
- Examples of Widget elements that are widely used in Android apps are shown
- Using Android Studio, developers can drag and drop these elements into the editor space and use them for app development.

## 6.5 Widget

---

For more clarity, the background area in this Figure represents the View, and individual buttons drawn on the View are Widgets.



## 6.5 Widget

- For more information on the View and Widget classes and the relationship between them see the View class
- Some of the classes are highlighted to show that Widget classes are subclasses of the View class
- Each Widget class has its unique attributes and usages

### View

public class View

extends [Object](#) implements [Drawable.Callback](#), [KeyEvent.Callback](#), [AccessibilityEventSource](#)

[java.lang.Object](#)



[android.view.View](#)

Known direct subclasses

[AnalogClock](#), [ImageView](#), [KeyboardView](#), [MediaRouteButton](#), [ProgressBar](#), [Space](#), [SurfaceView](#), [TextView](#), [TextureView](#), [ViewGroup](#), [ViewStub](#)

Known indirect subclasses

[AbsListView](#), [AbsSeekBar](#), [AbsSpinner](#), [AbsoluteLayout](#), [ActionMenuView](#), [AdapterView<T>](#) extends [Adapter](#), [AdapterViewAnimator](#), [AdapterViewFlipper](#), [AppWidgetHostView](#), [AutoCompleteTextView](#), [Button](#), [CalendarView](#), and 52 others.

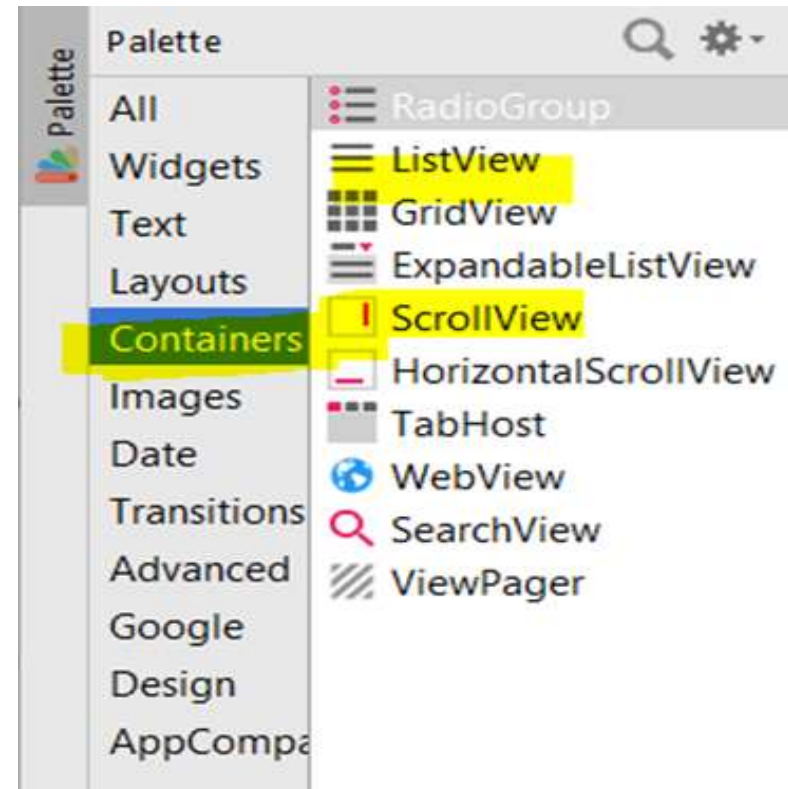
## 6.6 ViewGroup

- Another important Android UI class is ViewGroup
- ViewGroup objects are **invisible containers** that can contain other Views and ViewGroups
- The Views that reside inside a ViewGroup are called a child View
- ViewGroups are abstract classes and cannot be instantiated

## 6.6 ViewGroup

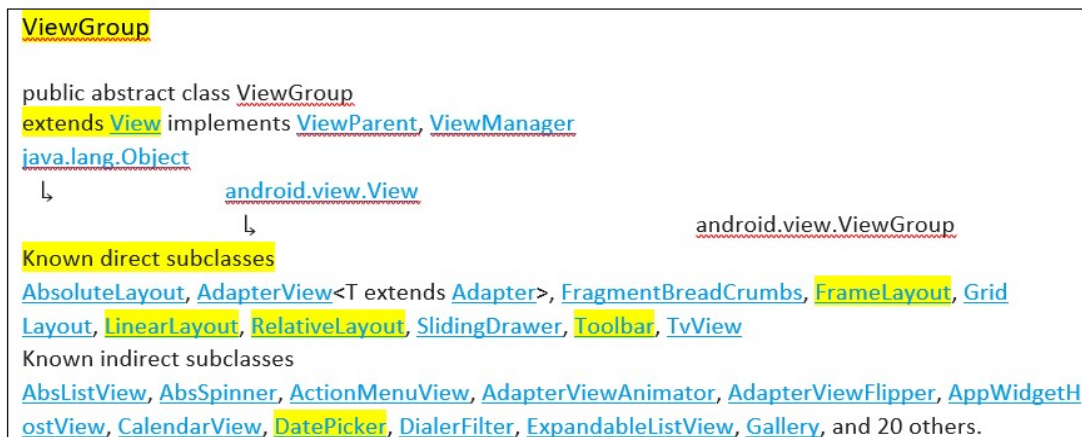
ViewGroups have many subclasses that can be instantiated.

Examples of ViewGroups include ScrollView, ListView, and RadioGroup.

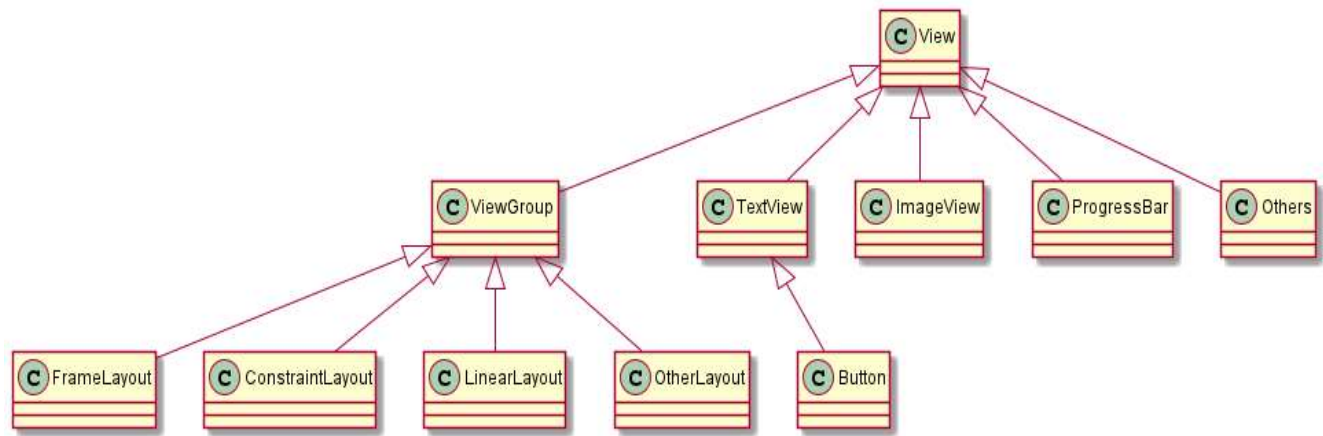


## 6.6 ViewGroup

- A common ViewGroup class are **Layout classes**.
- ViewGroup is a base class for Layout classes such as *Linear Layout*, *RelativeLayout*, *FrameLayout*, etc.
- It is used to organize and control the layout of a screen.
- The relationship between Views, ViewGroups, and Layouts is shown. class



## 6.6 ViewGroup



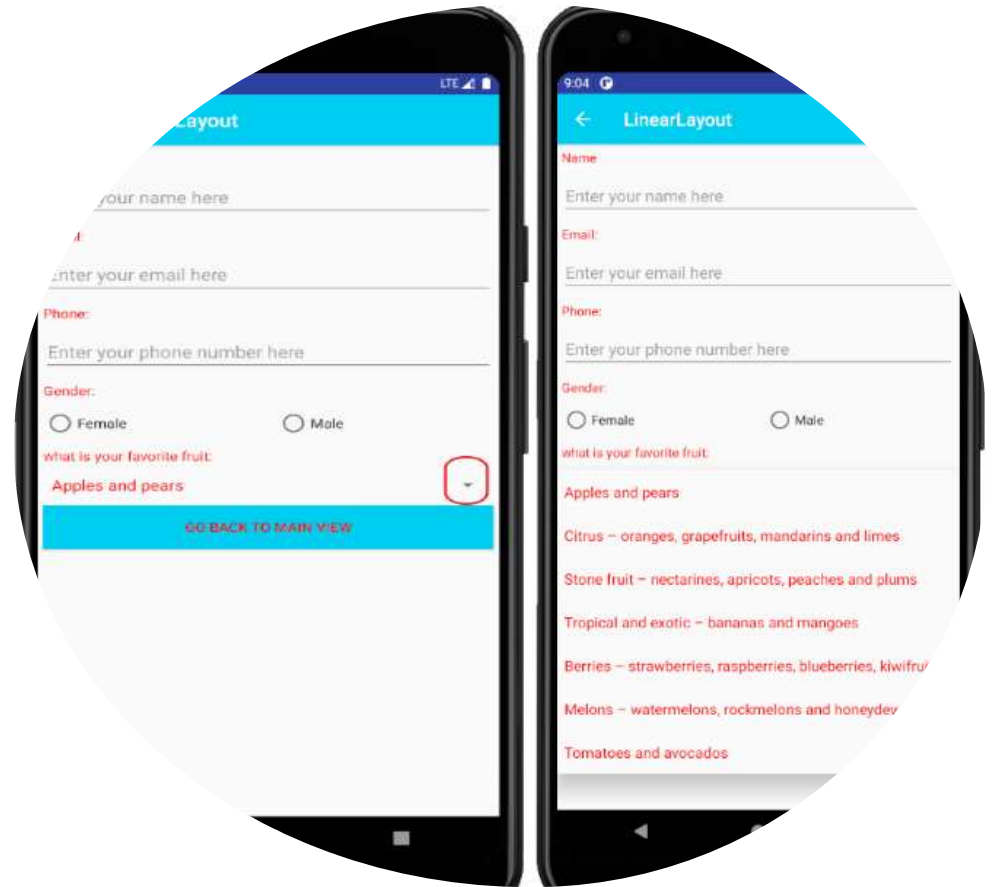
PlantUML diagram generated by SketchIt! (<https://bitbucket.org/pmesmeur/sketch.it>)

- A partial illustration of the View class hierarchy is shown in this figure:
- The figure shows that buttons are subclasses of TextView and are Views.
- It also shows that Layouts are ViewGroups and are Views as well.
- Understanding these relationships will help you in app development and in reusing code



## 6.6 ViewGroup

- In this example, the user interface is a simple linear vertical layout holding multiple Widgets.
- These include TextViews, EditTexts, RadioButtons, a drop-down list (i.e., a Spinner where users can select an item), and a button with the text *Go Back to Main View*.
- The app layout container is arranging Widgets on the surface, in this case, they are arranged linearly and in a vertical orientation.



## 6.7 App Layout

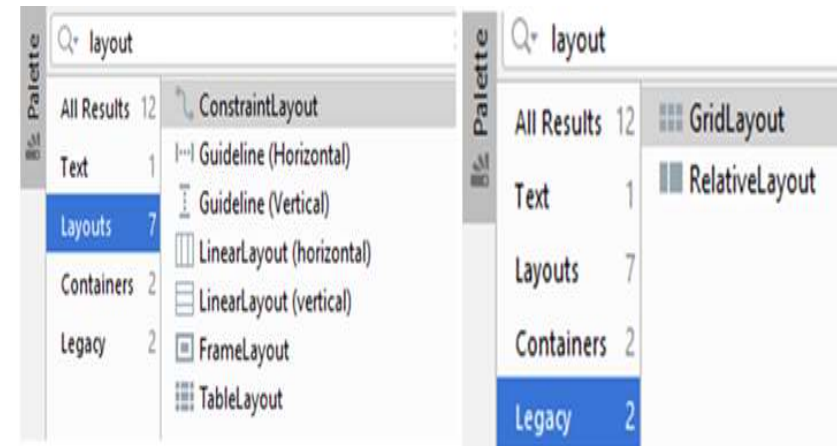
---

- Install and run the demo app, LayoutApplication.apk; You can download it from the book webpage.
- apk stands for Android application package, it is the app's executable code.
- Once you install LayoutApplication, you should see six buttons on its start screen.
- Check each Activity and its corresponding Layouts and resource files.



## 6.7 App Layout

- Some of the layout types that Android support are depicted in the figure
- Both *GridLayout* and *RelativeLayout* are considered legacy layouts, their use is not encouraged anymore.
- Android has introduced a more powerful layout, i.e., *ConstraintLayout*.



## 6.7 App Layout

- When you create your app, you can have nested layouts.
- You should limit the nesting and keep the hierarchy flat with a minimal number of Views and ViewGroups.
- The arrangement of the View hierarchy and the number of Views in the app has an impact on the app's performance.

## Part 2: Writing XML Layouts

---

- A layout is a ViewGroup object, i.e., an invisible container object that defines which part of an Activity screen will be occupied by which View or Widget objects, such as Button, TextView, EditText, or another Layout object such as FrameLayout or ConstraintLayout
- In Android, the user interface objects can be created using XML tags, programmatically at run time, or a combination of both
- When a combination of both approaches is used, you create your View and ViewGroup objects using XML elements and modify them at run time in your code

## 6.8 Declare UI elements in XML

- You can use XML files to create UI and Layout objects in your app
- When you use XML files to create UI objects and Layouts for your Activities, the name of the UI class will become an element in the XML file
- Both the Button and TextView will then become elements inside the LinearLayout root element, i.e., they will become child elements of the root element

## 6.8 Declare UI elements in XML

### <LinearLayout

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
xmlns:tools="http://schemas.android.com/tools"
```

```
android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
```

```
android:background="@color/maroon"
```

```
android:orientation="vertical"
```

```
tools:context=".MainActivity">
```

### <Button

```
android:id="@+id/button_linear_layout"
```

```
android:layout_width="match_parent"
```

```
android:layout_height="wrap_content"
```

```
android:layout_margin="10dp"
```

```
android:onClick="onLinearLayoutClicked"
```

```
android:text="@string/button_linear_layout" />
```

### <Button

```
android:id="@+id/button_relative_layout"
```

```
android:layout_width="match_parent"
```

```
android:layout_height="wrap_content"
```

```
android:layout_margin="10dp"
```

```
android:onClick="onRelativeLayoutClicked"
```

```
android:text="@string/button_relative_layout" />
```

...

To create a layout file that organizes your UI objects linearly with a Button and a TextView in it, you need to create an XML file where LinearLayout is the root element. Both the Button and TextView will then become elements inside the LinearLayout root element, i.e., they will become child elements of the root element.

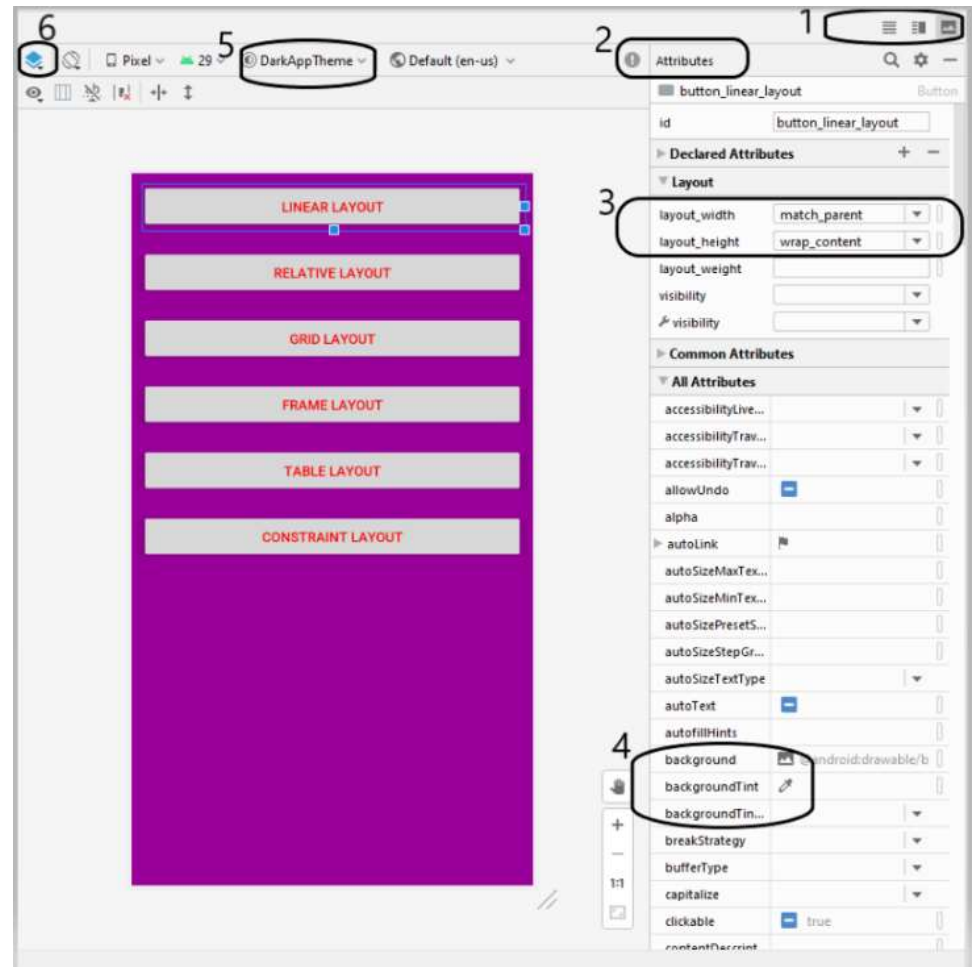
## 6.9 Android Studio's Layout Editor

Option 1: Toggle between layout code, layout design, or split screen to show code and design

•Option 2: Set widget attributes such as width, length, background colour, etc. Examples of such settings are numbered 3 and 4.

•Option 5: Set the app's theme

•Option 6: Select the design surface to show the design, the blueprint, or both





## 6.10 Defining UI Programmatically

```
public class MainLayoutActivity extends AppCompatActivity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        LinearLayout linearLayout = new LinearLayout(this);  
        // specifying vertical orientation  
        linearLayout.setOrientation(LinearLayout.VERTICAL);  
  
        // creating LayoutParams  
        LinearLayout.LayoutParams linearLayoutParam = new  
            LinearLayout.LayoutParams  
            (LinearLayout.LayoutParams.MATCH_PARENT,  
                LinearLayout.LayoutParams.MATCH_PARENT);  
        // set LinearLayout as a root element of the screen  
        setContentView(linearLayout, linearLayoutParam);  
    }  
    ...  
}
```

You can define your UI Layout and other UI components programmatically in your code

One of the fundamental principles of good programming is the separation of code from view

Using XML files to draw your app screen layouts and define your View objects using the XML element, makes it easy to provide different layouts for different screen sizes and orientations

## 6.11 LinearLayout Java Class



LinearLayout is a Java class that arranges its children's Views either horizontally in a single column or vertically in a single row



LinearLayout, and other Layouts that we will discuss later in this lesson, can be dealt with as regular Java classes



These classes have constructors, public fields, and public and protected methods, and inherit methods from their base classes

## 6.11 LinearLayout Java Class

For example, LinearLayout has four different constructors

---

`LinearLayout(Context context)`

---

`LinearLayout(Context context, AttributeSet attrs)`

---

`LinearLayout(Context context, AttributeSet attrs, int defStyleAttr)`

---

`LinearLayout(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes)`

<code>defStyleAttr</code>	int: An attribute in the current theme
<code>defStyleRes</code>	int: A resource identifier of a style resource

## 6.11 LinearLayout Java Class

The LinearLayout class also has multiple useful methods for setting and retrieving Layout properties.

<code>int getGravity()</code>	Returns the current gravity.
<code>int getOrientation()</code>	Returns the current orientation.
<code>void setGravity(int gravity)</code>	Sets the positions of the child Views; defaults to GRAVITY_TOP.
<code>void setHorizontalGravity(int horizontalGravity)</code>	Controls the alignment of the children.
<code>void setOrientation(int orientation)</code>	Sets the orientation, and the layout can be a column or a row.
<code>void setShowDividers(int showDividers)</code>	Sets how dividers should be shown between items in this layout.

## 6.11 LinearLayout Java Class

When you use XML files to represent classes, for every class constant and field there is a corresponding XML attribute.

XML attribute <LinearLayout>	Class LinearLayout fields
android: divider	Divider
android: gravity	Gravity
android: orientation	Orientation

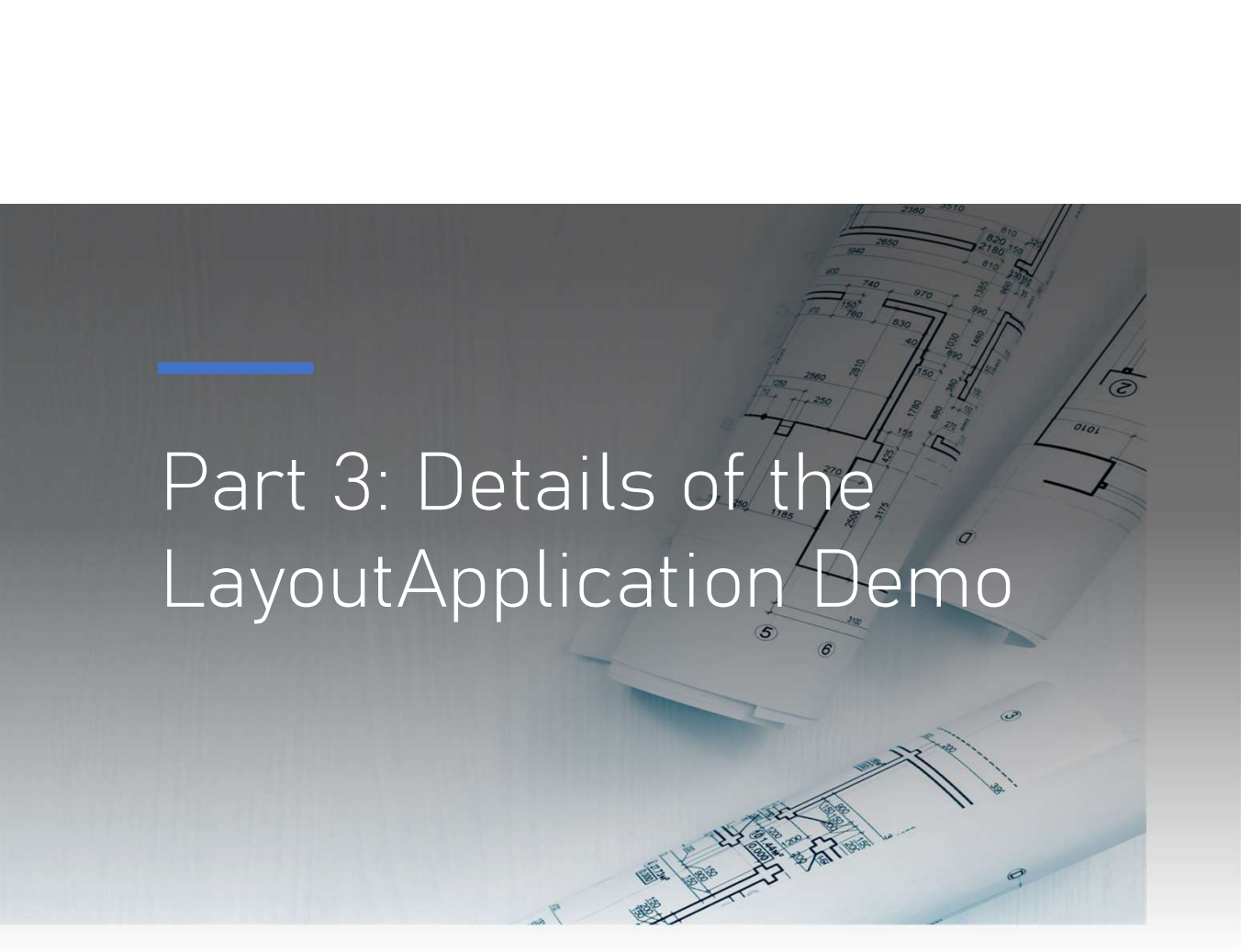
## 6.12 LayoutParams Java Class

- Another important Android class for defining UI programmatically is LayoutParams
- The LayoutParams class is used by Layout to tell Activities how they want to be set on the content view
- an Activity class can use LayoutParams like this:
- **setContentView(LinearLayout, LayoutParam);**

# LayoutParams

---

- The LayoutParams base class has two fields, **MATCH\_PARENT** and **WRAP\_CONTENT** that can be used with width and height dimensions
- MATCH\_PARENT sets the View to be as big as its parent and WRAP\_CONTENT sets the View to be just big enough to enclose its content
- There are other classes of LayoutParams for various subclasses of ViewGroups
- These classes extend the base LayoutParams class and have additional fields of their own



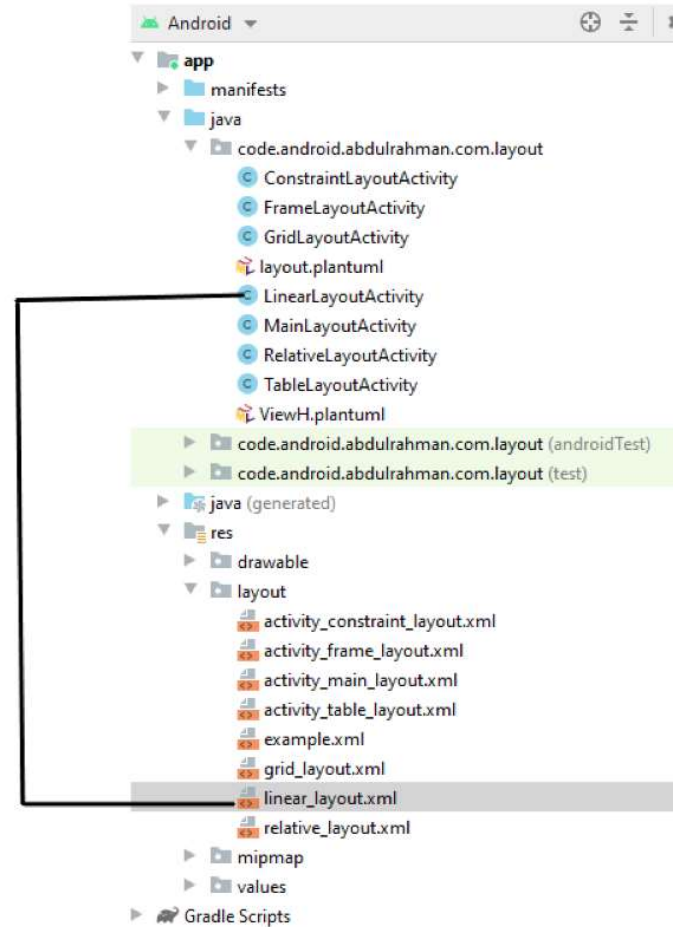
## Part 3: Details of the LayoutApplication Demo

- In this part, to demonstrate how you can use a Layout in your app, we go through the LayoutApplication code created for this chapter



## 6.13 MainActivity Layout

- For each Activity there is a corresponding Layout
- Even though you can create an Activity that is not a screen and that doesn't have a Layout, it is hardly used this way



## 6.13 MainActivity Layout

```
<LinearLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

```
xmlns:tools="http://schemas.android.com/tools"
```

```
android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
```

```
android:background="@color/maroon"
```

```
android:orientation="vertical"
```

```
tools:context=".MainActivity">
```

```
<Button
```

```
android:id="@+id/button_linear_layout"
```

```
android:layout_width="match_parent"
```

```
android:layout_height="wrap_content"
```

```
android:layout_margin="10dp"
```

```
android:onClick="onLinearLayoutClicked"
```

```
android:text="@string/button_linear_layout" />
```

```
<Button
```

```
android:id="@+id/button_relative_layout"
```

```
android:layout_width="match_parent"
```

```
android:layout_height="wrap_content"
```

```
android:layout_margin="10dp"
```

```
android:onClick="onRelativeLayoutClicked"
```

```
android:text="@string/button_relative_layout" />
```

- The XML snippet show is the activity\_main\_layout.xml which is associated with the MainActivityLayout Java code
- The layout is a simple linear layout with a root element and vertical orientation
- The property **tools:context=".MainActivity"** tells us that this Layout is associated with the MainActivity

## 6.13 MainActivity Layout

```
public class MainLayoutActivity extends AppCompatActivity {  
    ...  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main_layout);  
        Toast.makeText(getApplicationContext(),  
            getString(R.string.i_am_here_message),  
            Toast.LENGTH_LONG).show();  
    }  
    ...  
}
```

The MainLayoutActivity.java class loads the activity\_main\_layout.xml in the onCreate callback method using this line of code  
**setContentView(R.layout.activity\_main\_layout)**

## 6.16 Using Android Studio Design Option

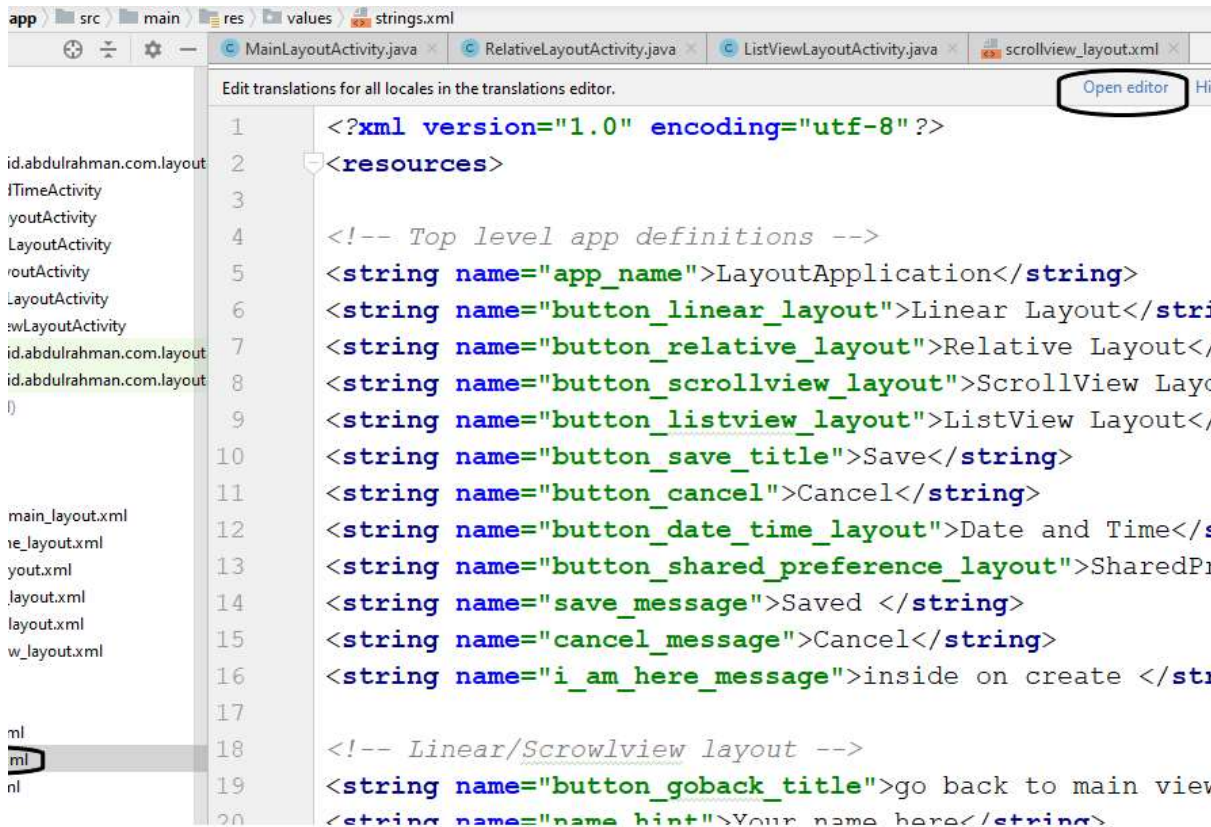
---

- you can use Android Layout Editor to create layouts for your Activities
- The editor provides three options to develop your Activity layouts:
  - the coding mode,
  - the design mode
  - a combination of both
- **For example, the properties of a Button object can be set by**
  - first dragging and dropping the button inside the LinearLayout root element,
  - highlighting the button,
  - and left-clicking to set the button properties such as  
Layout\_width, layout\_height, id, text, and the onClick method

## 6.17 Strings.xml File

- You can use a few HTML tags inside your strings.xml file to declare strings
- These are **<b>**, **<u>** and **<i>** HTML tags for bold, underline, and italics. All other HTML tags are ignored.
- Another tag that you can use is **\n** to start a new line or paragraph.
- Similarly, to write quotations, apostrophes, and any non-ASCII characters, use a backslash (**\**" , **\**' , **\é**).

## 6.18 String Editor



- If you open strings.xml in Android Studio and click on the open tab as shown below in Fig
- Using the Resource editor is very handy if you need to support multiple languages
- You can use the Resource editor to provide translations to the texts in the code

## 6.19 String Resources

- The strings.xml file can hold a wide set of objects
- These include:
  - strings,
  - arrays of strings,
  - integer arrays,
  - colors, and
  - styles/themes
- We widely use strings, string arrays, and integer arrays for our applications

## 6.20 RelativeLayout

---

- The RelativeLayout container class enables you to specify how child views are positioned relative to one another or the container
- The position of each View object can be specified as relative to sibling elements using properties such as  
left-of, right\_of, top, or below another View
- You can also position an element **relative to the parent container area** using properties such as aligned to the  
bottom, left, or center of the container



## 6.20 RelativeLayout

---



RelativeLayout is a legacy class now. Its use is no longer encouraged



Currently, Android supports a more sophisticated Layout called **ConstraintLayout** which will be discussed later



In our Application Layout app, RelativeLayout uses a simple relative layout. We have positioned three objects relative to each other and the parent container.

## 6.20 RelativeLayout

---

- The *AnalogClock* has been put at the bottom of the layout using ***layout\_alignParentBottom***
- the WLU logo has been put at the top of the layout using *ImageView* and ***layout\_alignParentTop***,
- and the *RatingBar* is situated in between the two widgets automatically.

## 6.20 RelativeLayout

---

- You can explicitly position the RatingBar between the other two Widgets using the ***below*** and ***above*** layout attributes.
- You can also use **`android:paddingBottom`** and **`android:paddingTop`** to control the exact position of *RatingBar* or any UI object between the other two Widgets.
- RelativeLayout has other properties. These include: *match\_parent* which will fill the screen ; the *padding-left* and *padding-right* to position Widgets in the container.
- The code for the RelativeLayout button in our demo app is in RelativeLayoutActivity

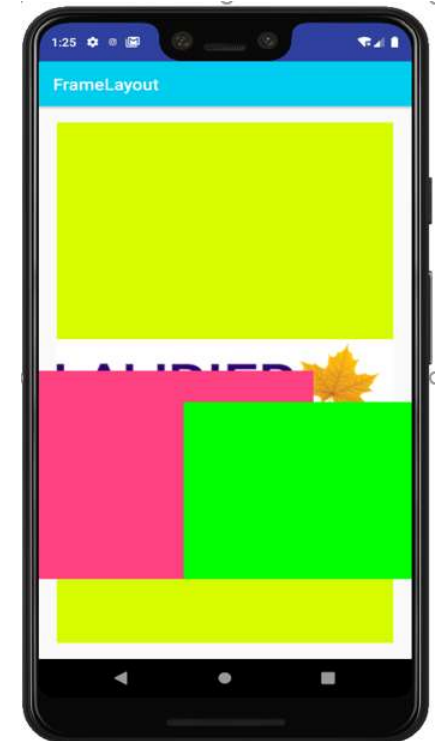
## 6.21 Other Layouts

- The `LayoutApplication` demo app includes other Layouts, and they are `FrameLayout`, `TableLayout`, `GridLayout`, `ConstraintLayout`
- For coding and layout implementation, see the source code of the `LayoutApplication` demo app

## 6.21.1 FrameLayout Layouts

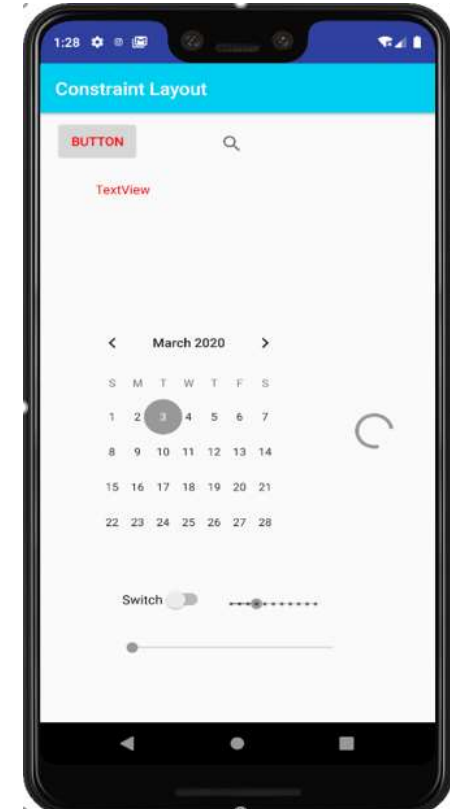
---

- Elements of *FrameLayout* are stacked on top of each other and the last element in the layout file is on the top of the stack.
- We can see this in where the last element, the green frame, is found on top of the others.
- FrameLayout is often used with the ***fragment*** View to draw on a part of the Layout.

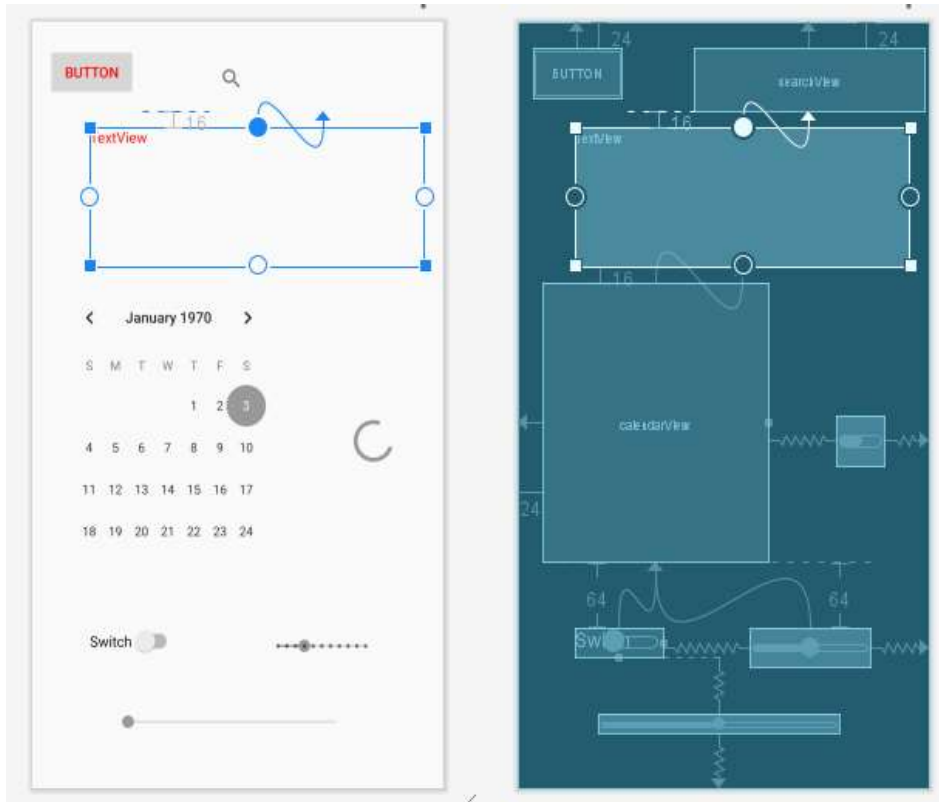


## 6.21.2 ConstraintLayout

- It works similarly to RelativeLayout except that it is more sophisticated and has more attributes.
- With ConstraintLayout, you can align and declare a precise position of an element relative to the other elements and the parent element in the Layout.
- It is a default Layout when you create an activity.

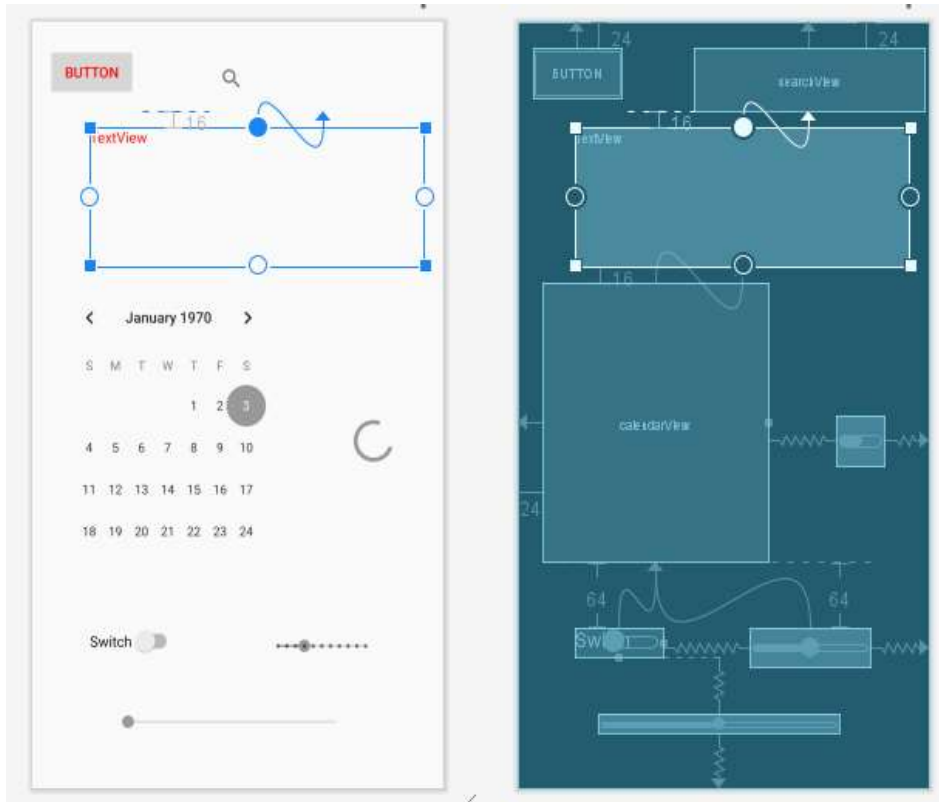


## 6.21.2 ConstraintLayout



- Using the Android Studio Layout Editor and ConstraintLayout, you can create complicated layouts easily.
- You can do that by simply connecting so-called **constraint connectors**.
- The figure shows an example of a complicated layout.

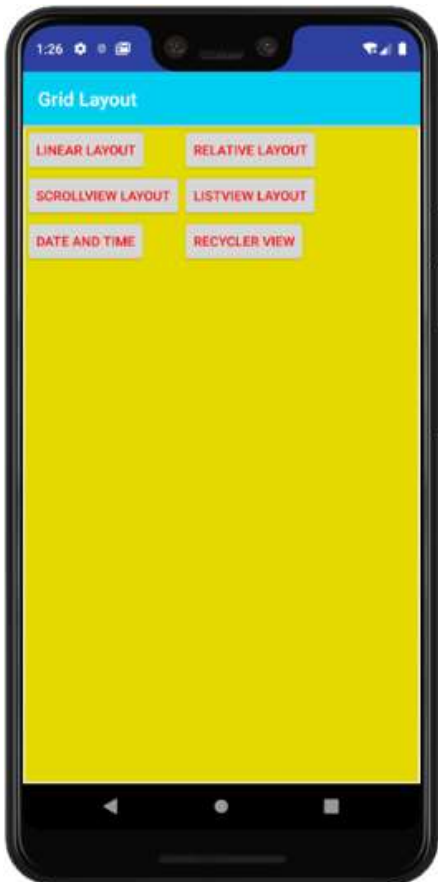
## 6.21.2 ConstraintLayout



- What is shown on the **right-hand side** of Fig. is called a layout view, or a **blueprint**, and what you see on **the left-hand side** is a design view.
- What you see in the design view is what you get in the final interface of the app.
- The constraint connectors are showing on the blueprint view.
- By connecting these connectors, you can define the position of Views relative to other elements and parent Views.



## 6.21.3 GridLayout



- The Grid layouts is shown in Fig. To create *GridLayout*, you have to declare **column** and **row** counts, see the code snippet

```
<GridLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:layout_margin="2dp"
android:background="@color/gold"
android:columnCount="2"
android:rowCount="3"
tools:context=".GridLayoutActivity"/>
```

## 6.21.3 TableLayout



- To create *TableLayout*, you have to declare rows, and you can have any number of columns in each row.
- The number of columns in each row does not have to be the same, and that makes it different from the Grid layout.

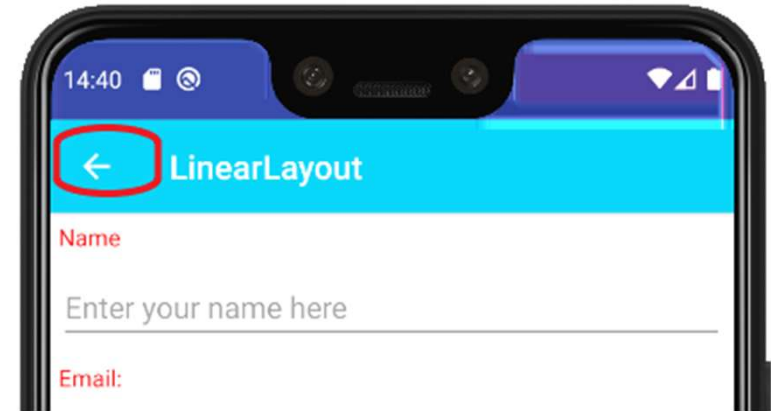
```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout

xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context="TableLayoutActivity">
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView ...
        ....
    </TableRow>
</TableLayout>
```

## 6.22 Parent-Child relationship between Activities

- To help navigation between app screens, you can **define parent/child relationships between Activities** in the manifest file.
- This results in creating an arrow on the app's menubar, and when pressed, it will end the current Activity and bring in the parent Activity.



## 6.22 Parent-Child relationship between Activities

---

- To define the parent/child relationship between Activities, add a new entry, i.e., a meta-data entry, to the application element of the manifest file.
- Inside the meta-data element, declare that the app supports the parent-child relationship and define which Activity is the parent Activity.
- This addition to the manifest file is shown below.

```
<application>  
...  
<meta-data  
    android:name="android.support.PARENT_ACTIVITY"  
    android:value="MainActivity" />  
</application>
```

## 6.22 Parent-Child relationship between Activities

- You also need to update the definition of all Activities declared inside the manifest file to tell which Activity is its parent Activity
- For example, in the statement below, we are declaring that the parent Activity of ConstraintLayoutActivity is MainActivity.

```
<activity android:name=".ConstraintLayoutActivity"  
    android:parentActivityName=".MainActivity" />
```

# Part 4: Styles, Themes, and Dimension

- Like CSS in HTML programming, the ***Android style*** specifies the visual properties of the elements that make up the app's interface.
- In the layout file, you define properties for individual Views on your screen.
- You can **create a style** to define properties for multiple Views of the same type in your app by **creating a file named style.xml inside the res folder**

# Part 4: Styles, Themes, and Dimension

if you use 20 buttons in your app, you can define a style that includes the height, padding, margins, font size, and font colours, i.e., a collection of attributes, that can be applied to all buttons

- This enables the reuse of property definitions and a consistent look throughout the app



## 6.23 Defining a Style File

To define a style:

- You give a name to your style
- inherit style properties from the existing styles, i.e., the parent styles that are defined in the Android API
- In the code sample below, the style name is “**LightAppTheme**”, and it inherits properties from an existing theme, “**Theme.AppCompat.Light.NoActionBar**”.

```
<resources>
  <style name=" LightAppTheme"
    parent="Theme.AppCompat.Light.DarkActionBar">
  </style>
</resources>
```



## 6.23 Defining A style File

- You customize or add properties to the style by **defining <item> entries** inside the style elements
- The name in each item is an Android attribute that you would otherwise use in the layout file to define a View attribute
- The <item> elements are name-value pairs.
- The value for <item> can be:
  - a keyword *string*,
  - a *hex color*, or
  - a reference to another resource type using the at-symbol “@”, e.g., @style.
- **You can define multiple styles in one file using the <style> tag, but each style should have a unique name.**

## 6.23 Defining A style File

```
<style name="forTextStyle"
    parent="Theme.AppCompat.Light.DarkActionBar">

    <item name="android:capitalize">characters</item>
    <item name="android:textSize">20sp</item>
    <item name="android:textAppearance"> ?android:textAppearanceLarge </item>
    <item name="android:buttonStyle">@style/Widget.AppCompat.Button.Borderless</item>
</style>
```

## 6.24 Applying Styles

- Depending on which style you refer to in your manifest file, your app's look will be different
- A style can be applied to an individual View when referenced inside an element definition in the layout file or to an entire Activity or application when referenced inside the manifest file
- We have defined a style called **MyTextViewStyle** for a TextView and referenced it inside the TextView definition in the layout file using `@style`
- We also created a new Activity called `UsingStyleActivity` and a new layout file called `activity_using_style` with a TextView element inside it

## 6.24 Applying Styles

Listing 6.14 A style definition that we reference inside a layout file.

```
<resources>
<style name="MyTextViewStyle">
    <item name="android:textColor">@color/blue</item>
    <item name="android:textStyle">bold</item>
    <item name="android:padding">10dp</item>
    <item name="android:inputType">textMultiLine</item>
    <item name="android:maxLines">100</item>
    <item name="android:scrollHorizontally">>false</item>
</style>
</resources>
```

Listing 6.15 A Layout file where style is referenced.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android=http://schemas.android.com/apk/res/android ... >
    <TextView
        android:id="@+id/textViewID"
        style="@style/MyTextViewStyle" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

## 6.23 Defining the App's Theme



FIG

CHAPTER 06: USER INTERFACE ESSENTIAL CLASSES, LAYOUTS,  
STYLES, THEMES, AND DIMENSIONS

## 6.23 Defining the App's Theme

- In our demo app, we referenced *LightAppTheme* in the manifest file.
- The app style in the manifest file is set like this:  
**`android:theme="@style/LightAppTheme"`**
- *LightAppTheme* is the name of a style that has been defined in the style file.
- The at-symbol “@” is an instruction to the Android system to find the style file in the *res* folder.

## 6.23 Defining the App's Theme

- You create an Android theme by defining a style file with attributes such as Colors, TextApperance, Dimens, Drawables, Shapes, Buttons Styles
- Attributes are not specific to any individual View type, but rather are applicable more broadly and referenced in the manifest file.
- For example, you can define a colorPrimary attribute and apply it as a background colour to all of your Activities.
- You can define textColor and TextApperance and apply them to all the texts in your app.

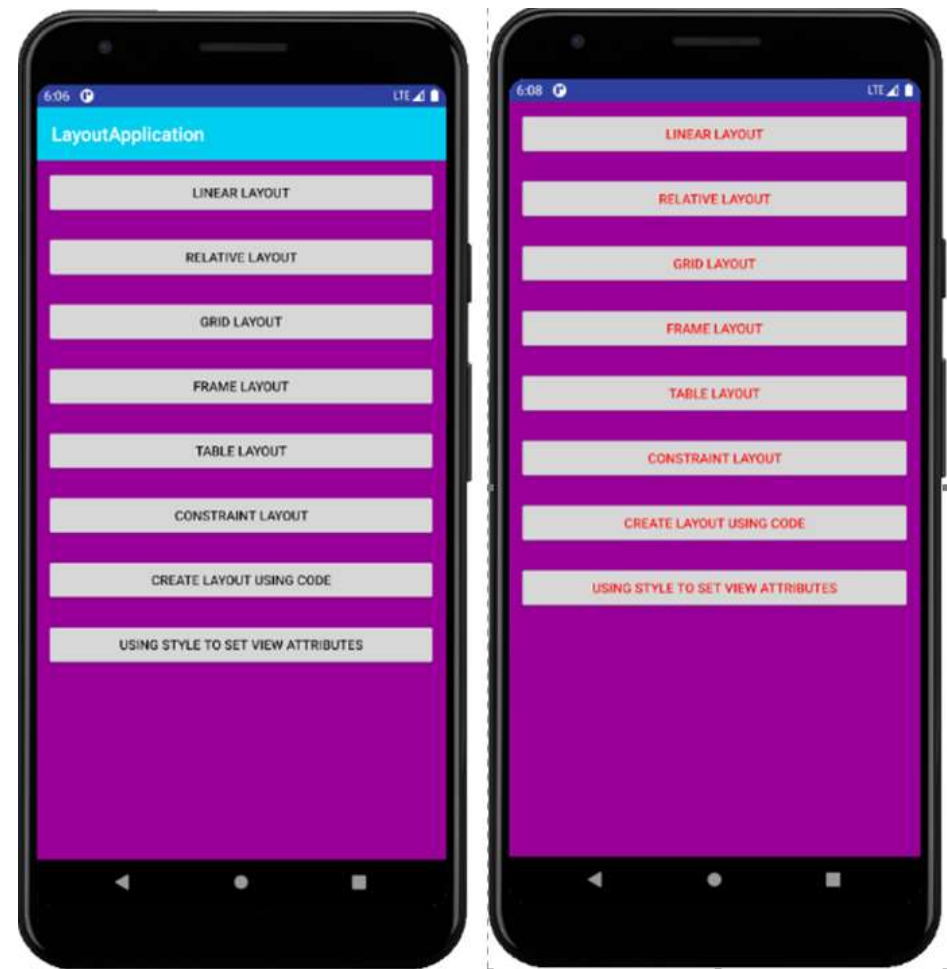
---

## 6.23 Defining the App's Theme

When designing a theme for your app, **start by selecting a theme that matches your needs.**

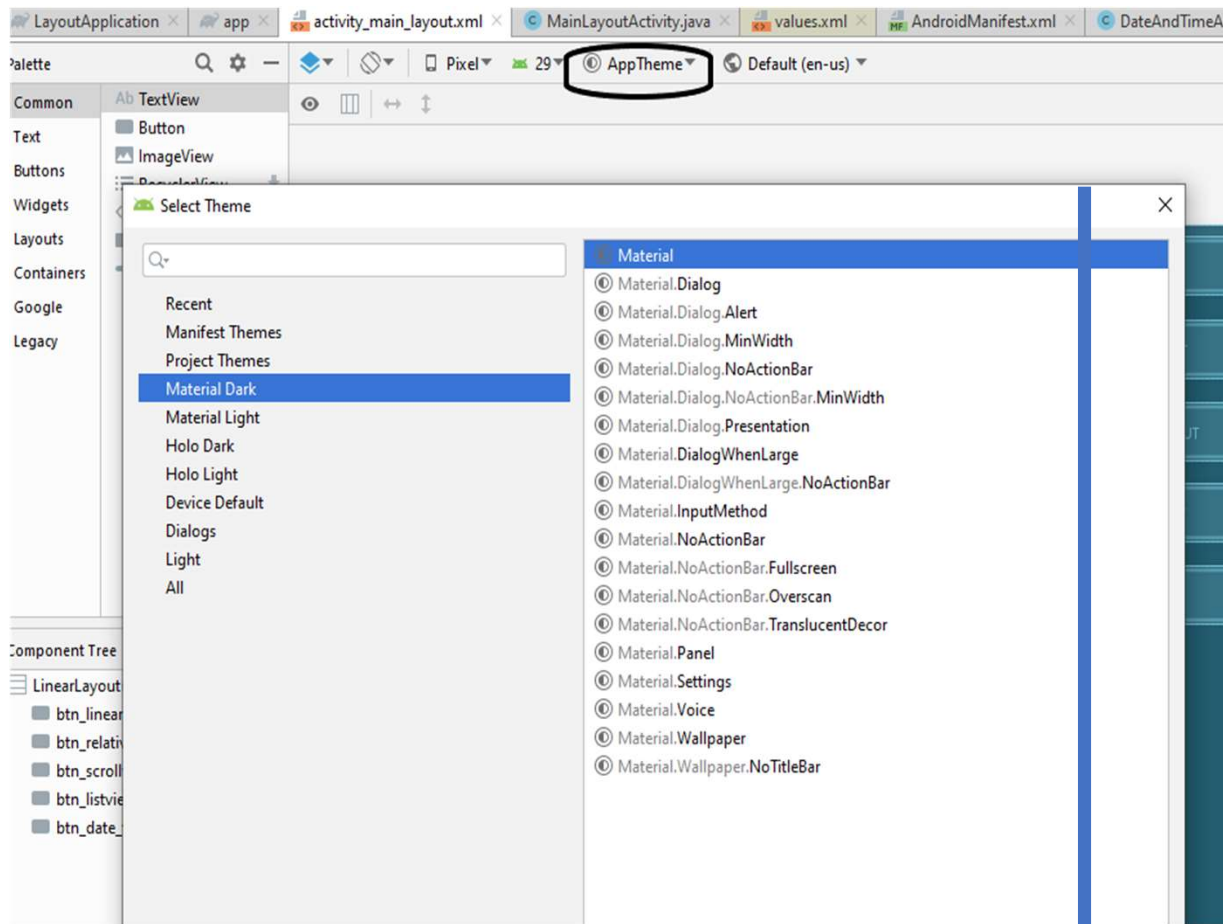
You can customize your theme later as you learn more about your app's requirements.

Fig. presents a snapshot of an app's appearance when it is run with two different themes.





## 6.25 The Difference Between a Theme and Style

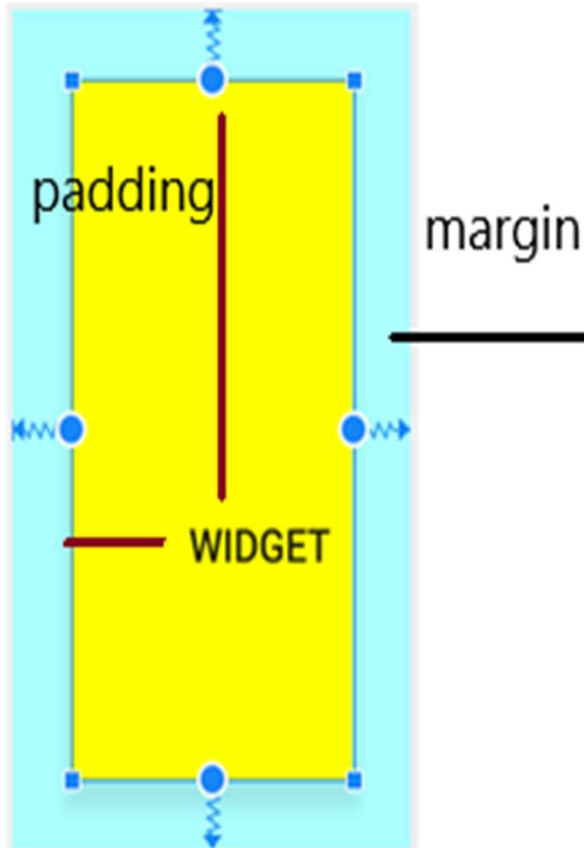


## 6.25 The Difference Between a Theme and Style

---

- A theme is a style applied to an entire application, instead of an individual View type
- When a style is applied as a theme, referenced inside the manifest file, every View in the application will apply each style property that it supports.
- Android provides multiple system themes that you can choose from when building your apps.
- Examples of such themes include.
  - Material (dark version)
  - Material Light (light version)
  - Material Light with NOActionBar

## 6.26 Padding and Margin View Properties



- View objects have other attributes
- The margin attribute names are `android:layout_marginBottom`, `android:layout_marginEnd`, `android:layout_marginLeft`, `android:layout_marginRight`, and `android:layout_marginTop`
- margin attributes define the outside space between the border and the other elements next to the current View
- padding is the inside space between the border and the actual View's content

## 6.27 Gravity and Weight View Properties

---

### *android:layout\_gravity.*

Defines how the child view should be placed or positioned, on both the X and Y axes, within its enclosing layout; some positions include *top*, *bottom*, *left* and *right*; *start*, *center*, and *end*.

```
android:layout_gravity="bottom|left"
```

```
android:layout_gravity="bottom|right"
```

---

### *android:layout\_weight.*

Specifies how much of the **extra space** in the linear layout is allocated to the View object that has this parameter. The value is between 0 and 1.

Zero space if the View should not be stretched.

Otherwise, the extra pixels will be distributed among all Views whose weight is greater than zero.

```
android:layout_weight="0.5"
```

## 6.28 Dimensions of a phone and UI



- **The screen size, both the width and height of a device, as well as the pixel density influences the design of your app**
- When designing your app's screen, it is helpful if you have information about the display characteristics of the device you are designing for
- The physical size of the phone is 5.5 inches diagonally, the resolution is 1440 x 2560 pixels, and the pixel density is 534 ppi where ppi is pixels per inch

# scaling units

Here are some of the scaling units that you need to be aware of when specifying the UI for your app.

- px – screen pixels.
- in – inches based on the physical screen size.
- mm – millimeters based on the physical screen size.
- pt – 1/72 of an inch based on physical screen size.
- **dp or dip – device-independent unit relative to a 160 dpi screen.**
- sp – similar to dp but used for font sizes.
- dpi – dots per inch; the number of pixels within a physical area of the screen is referred to as dpi.

# scaling units

Here are some of the scaling units that you need to be aware of when specifying the UI for your app.

Most people confuse dp with ppi (pixels per inch)

One dp is equivalent to one pixel on a 160 dpi ((dots per inch)) screen.

On a 160dpi screen,  $1dp == 1px == 160/160in$ ,  
but on a 240dpi screen,  $1dp == 240/160 = 1.5px$

## 6.28 Do it yourself

---

Run the demo app created for this lesson on different devices and study the XML layout look on both screens

---



## 6.5 Chapter summary

- In this Lesson, we studied how to create Views, Layouts, Widgets, and other components and their properties that enable you to create a nice-looking and user-friendly interface application.
- We studied the different types of Layouts that Android supports.
- We studied how to define your UI object using XML elements and XML files, as well as how to create Views and Widgets programmatically.
- Other topics studied in this lesson include phone styles, themes, and phone dimensions.
- We will continue with the user interface in the next chapter to study  
ListView, ScrollView, RecyclerView, Pickers, and more.

# Check Your Knowledge

These are some of the fundamental concepts and vocabularies that have been covered in this chapter. To test your knowledge and your understating of this chapter, you should be able to describe each of the below concepts in one or two sentences.

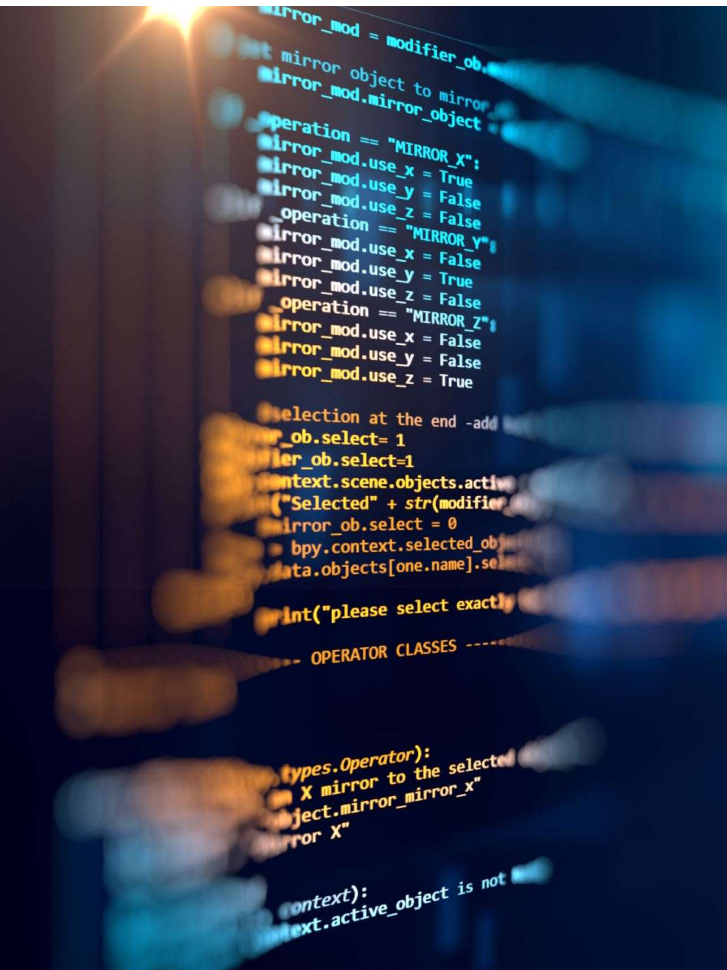
- `android:layout_weight`
- Button
- Checkbox
- Constraint Layout
- dp
- Frame Layout
- Grid Layout
- Linear Layout
- Margin
- Padding



# Check Your Knowledge

These are some of the fundamental concepts and vocabularies that have been covered in this chapter. To test your knowledge and your understating of this chapter, you should be able to describe each of the below concepts in one or two sentences.

- Radiobutton
- RadioGroup
- Relative Layout
- Spinner
- Style
- Table Layout
- TextEditor
- TextView
- Theme
- etc



## Further Reading

- Build a UI with Layout Editor,  
available <https://developer.android.com/studio/write/layout-editor>
- ConstraintLayout,  
available <https://developer.android.com/reference/android/support/constraint/ConstraintLayout>
- Design app themes with Theme Editor,  
available <https://developer.android.com/studio/write/theme-editor>
- Layouts,  
available <https://developer.android.com/guide/topics/ui/declaring-layout>
- Linear Layout,  
available <https://developer.android.com/guide/topics/ui/layout/linear>
- Relative Layout,  
available <https://developer.android.com/guide/topics/ui/layout/relative>