Lesson 6: Android Layouts, Styles, Theme and Menus

Introduction

In the previous lessons, you were introduced to various View types available in Android framework. They were only a subset of a large number of View classes also called widgets which are available for user interactions. Each of these views has a default behavior and an interface. However, sometimes your application might require slightly different views from what's available in Android SDK.

This lesson starts by explaining views and layouts that show you how to create your own reusable customized views.

Views

A View in Android development is the basic component for the app user-interface. In Android SDK, the class is called View. It is the highest-level class for any user interface component or widget you may use in your applications. All user interface widgets that you have seen before like Button, TextView, and others are subclasses of class View.

Layouts

A layout defines the visual structure for a user interface, such as the user interface for an activity or app widget. You can declare a layout in two ways:

1-Declare user elements in XML: Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.

For example the following XML code is used to declare a button in the activity layout:

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

Later, in this lesson you will learn how you can control the layout format using the XML code.

2- Instantiate layout elements at runtime: Your application can create `View` and `ViewGroup` objects (and manipulate their properties) programmatically such as creating a dynamic button using Kotlin code. This method is not recommended because it takes more time and effort.

Layout Types

You may use different types of layouts in your app design. First, you create a new Android project, and then demonstrate the different types of layouts to determine which one suits the most in your app graphical interface. To know more about these types, you should create a new Android project, then test each type of these layouts.

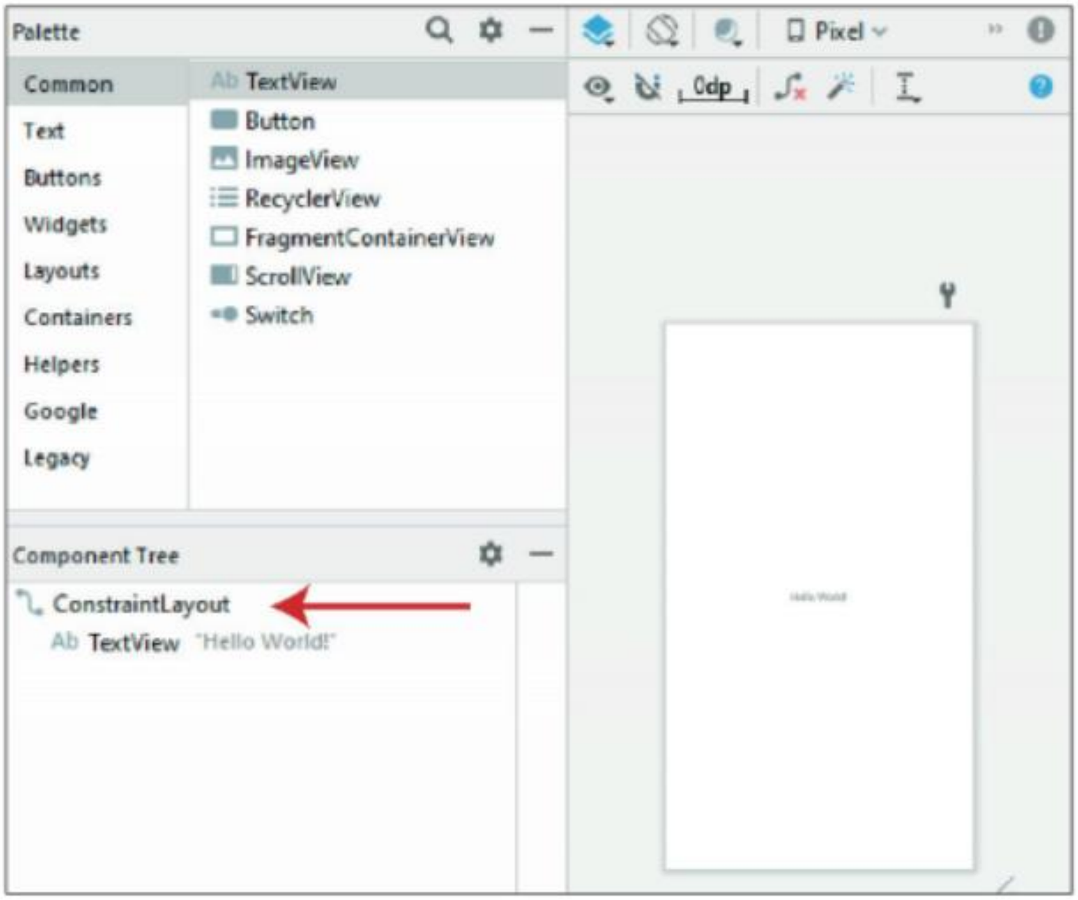
To create a new Android project, follow the steps below:

1-Open Android Studio, and then click **File → New → New Project.**

2- Select **Empty Activity**, and click **Next.**

3- Enter the application Name: **Lesson06 and** then click **Finish**.

The following figure displays the default **activity_main.xml** (app → res → layout) file content which also includes the view of your app layout. The component tree panel displays the default layout type of the activity which was created when you first built your app. The default layout type is: **ConstraintLayout** as illustrated in the figure below:



Constraint Layout:

Constraint Layout is a powerful new layout engine released as a part of Constraint Layout library in Android Studio 2.2 and later versions. It is fully integrated with Android Studio's layout editor so you can build the full layout without the need to edit the XML manually.

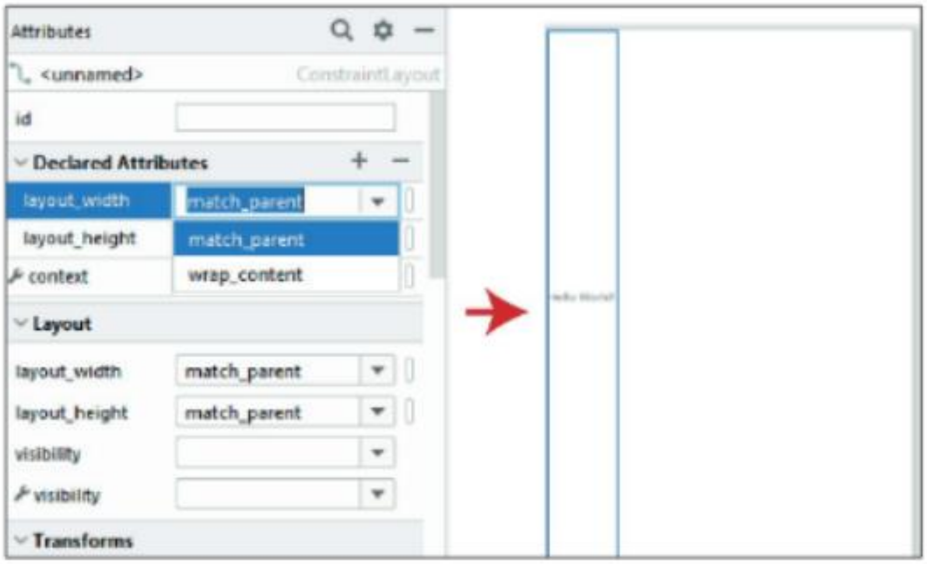
This layout allows us to specify constraints (Margins) that would decide the position of each sub element such as text, button, radio box, check box or other widgets within the layout.

3- Open the **activity_main.xml** in the Code mode, and you should get the following XML code:

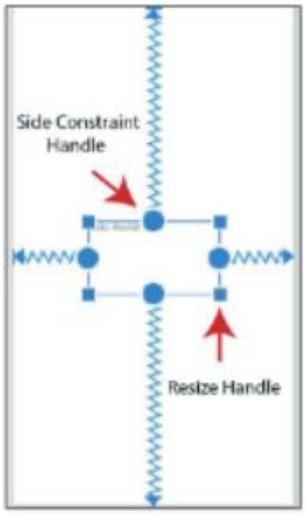
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

The gray highlighted parts of the previous code, which include **android:layout_width** and **android:layout_height** attributes, determine the height and width of your activity layout. The "**match_parent**" attribute value means that this layout will cover the entire phone screen (height and width).

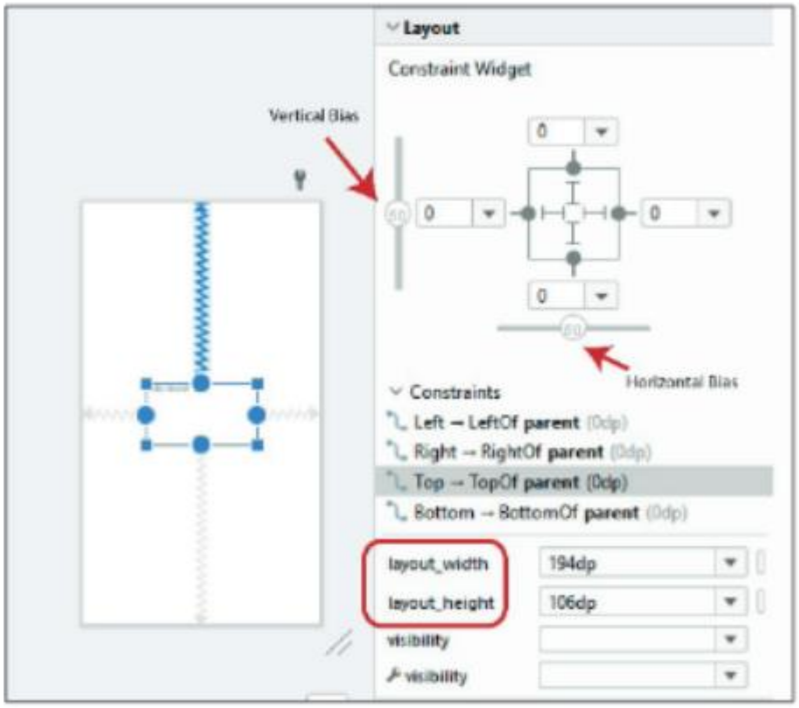
4- Click "**Design**" button to return to the design mode of your activity layout so you can change your layout attributes easily. As illustrated in the figure below, if you change the attribute value of "**layout_width**" to "**wrap_content**", the layout width will change to take the width of the contents, as illustrated in the following figures:


Attributes
<unnamed> ConstraintLayout
id
Declared Attributes
layout_width: match_parent
layout_height: match_parent
context: wrap_content
Layout
layout_width: match_parent
layout_height: match_parent
visibility
visibility
Transforms

Every **View** within a Constraint Layout has constraint handles which allow positioning views relative to other widgets. The following figure displays the resize handle which allows you to resize the widget and the side constraint handle which lets you specify the location of the widget relative to other widgets or the device edge.



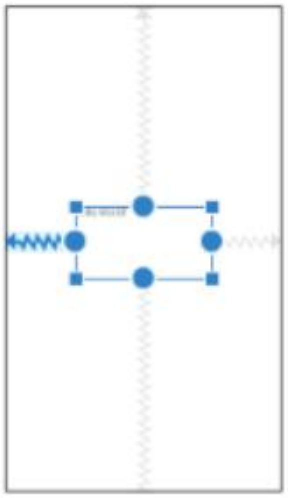
You can also use the vertical and horizontal biases in the **Attributes → Layout** tab as illustrated in the figure below to determine how a view should be positioned along the vertical and horizontal axis using bias values. Also, check the effect of change of values of the **layout_width** and **layout_height** of this view.



Left → LeftOf **parent** (0dp)
Right → RightOf **parent** (0dp)
Top → TopOf **parent** (0dp)
Bottom → BottomOf **parent** (0dp)

layout_width	194dp
layout_height	106dp
visibility	
visibility	

Click on the blue circle of any widget that has a defined constraint, then as you see in the figure below, if you press **Delete**, you will delete this constraint, or you may right click and get a shortcut menu that has a lot of choices related to this constraint.



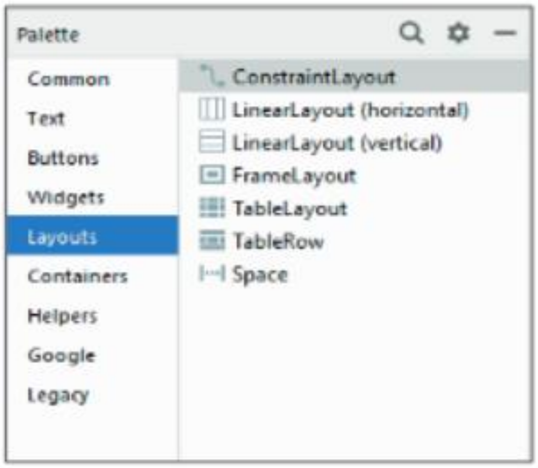
Linear Layout

A Linear Layout is a layout that arranges other views either horizontally in a single column or vertically in a single row. This layout is a view group used to align all children (widgets) in a single direction, vertically or horizontally.

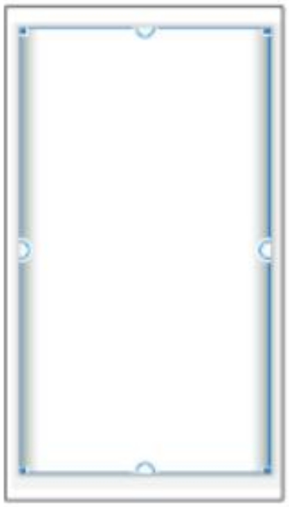
If you want to add more than one widget such as three radio buttons or more, it is a good idea to arrange them in a **LinearLayout** (horizontal) as children widgets or if you want to display them in one row or in a **LinearLayout** (vertical), then you have to display them in the same column. This method guarantees you that they will appear in the same shape even though the device screen resolution or size change. Also, that will give you more control on the format of this part of the interface.

You may make the **LinearLayout** a child of the main **ConstraintLayout** as you will see in the next example:

Continue using the same previous Android project (Lesson06). Click the **Layouts** on the palette panel to get all layout types which can be used to design your app activities, as illustrated in the following figure:



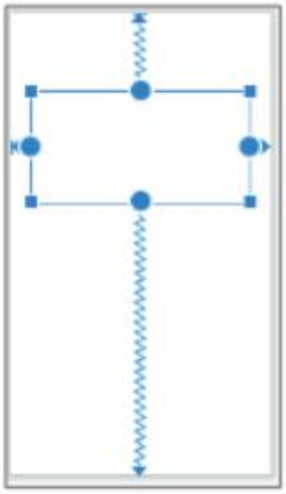
1- Delete the **TextView** "Hello World" from your activity and then drag and drop "**LinearLayout (horizontal)**" to your activity. You will get the following figure:



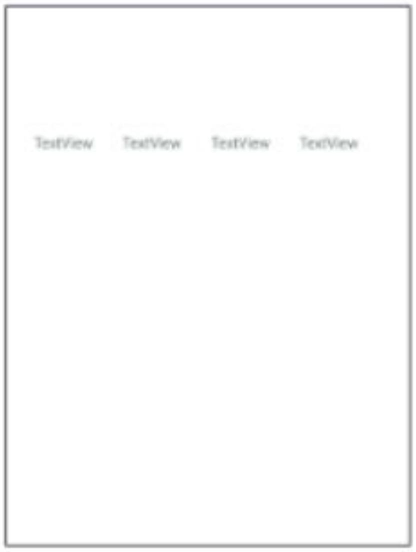
2- You should resize and add constraints for this Liner layout to be as follows:

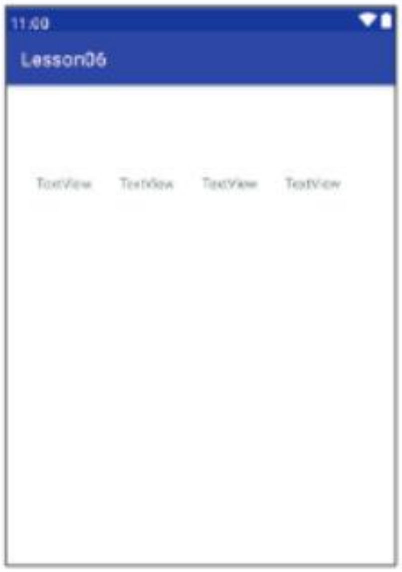
/

Recopilado por SLMG

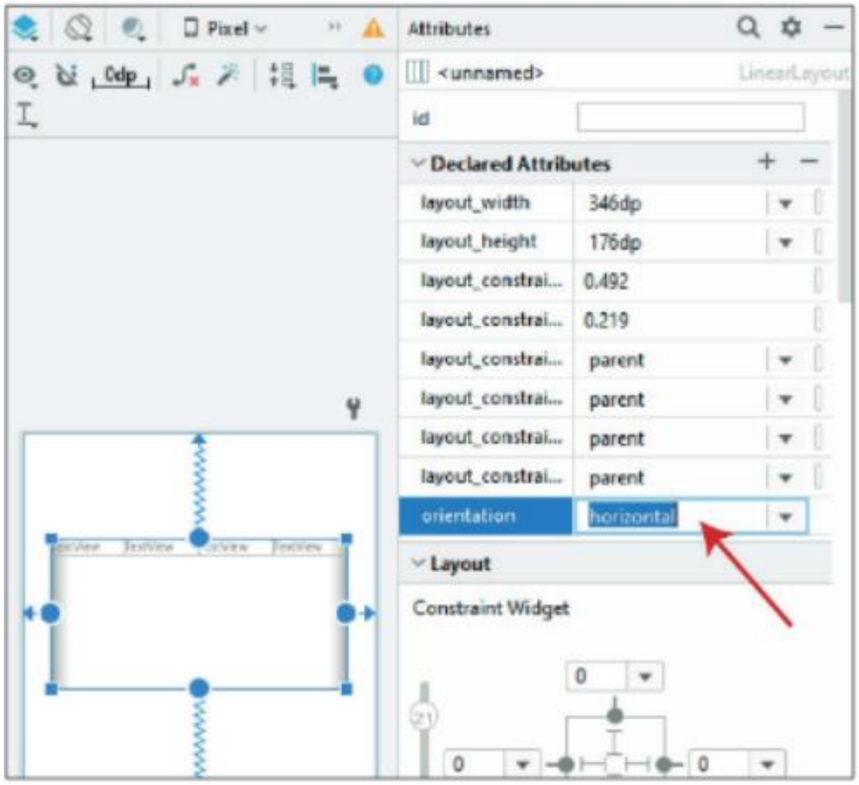


3- Add four **TextView** widgets to your **LinearLayout** in the **Design** mode using drag and drop technique. You will get the following figure:





4- Select the **LinerLayout**, click the **Attributes** panel, and as illustrated in the figure below, change the orientation attribute value to vertical, then all the children widgets (**TextViews**) will be arranged vertically.



id	
layout_width	346dp
layout_height	176dp
layout_constrai...	0.492
layout_constrai...	0.219
layout_constrai...	parent
layout_constrai...	parent
layout_constrai...	parent
layout_constrai...	parent
orientation	horizontal

If you open the **activity_main.xml** file in the **Code** mode, you will see that you may change all these **LinearLayout** attributes, and this **LinearLayout** is a child of the main **ConstraintLayout**.

Note: What would happen if you removed the constraint layout from the activity_main.xml file and used only linear layout in your layout design? You wouldn't be able to move the widgets (texts, buttons and others) easily on your activity as you would do with the constraint layout.

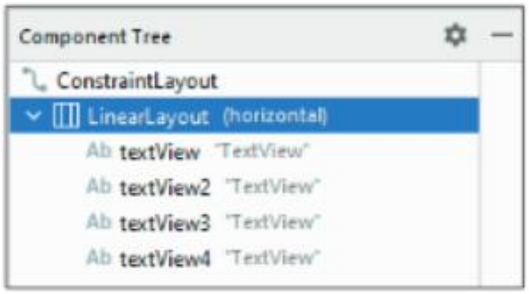
Relative Layout

RelativeLayout arranges child views (widgets) in relative positions to each other. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent **RelativeLayout** area (such as aligned to the bottom, left or center).

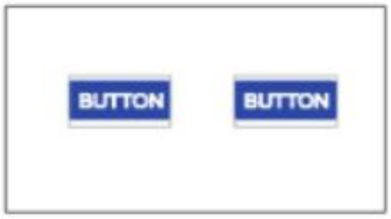
For example, you may position a button on the layout to the left, right or on top of another button.

Example: Use the same previous Android project and follow the following steps:

1- Delete all existing widgets. If you still have a **LinearLayout**, select the **LinearLayout** in the "**Component Tree**", press **Delete** and keep only the **ConstraintLayout** in your main activity as illustrated below:



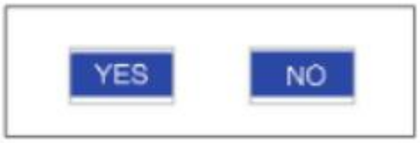
2- Add two buttons to your activity using drag and drop technique and keep them next to each other as illustrated in the following figure:



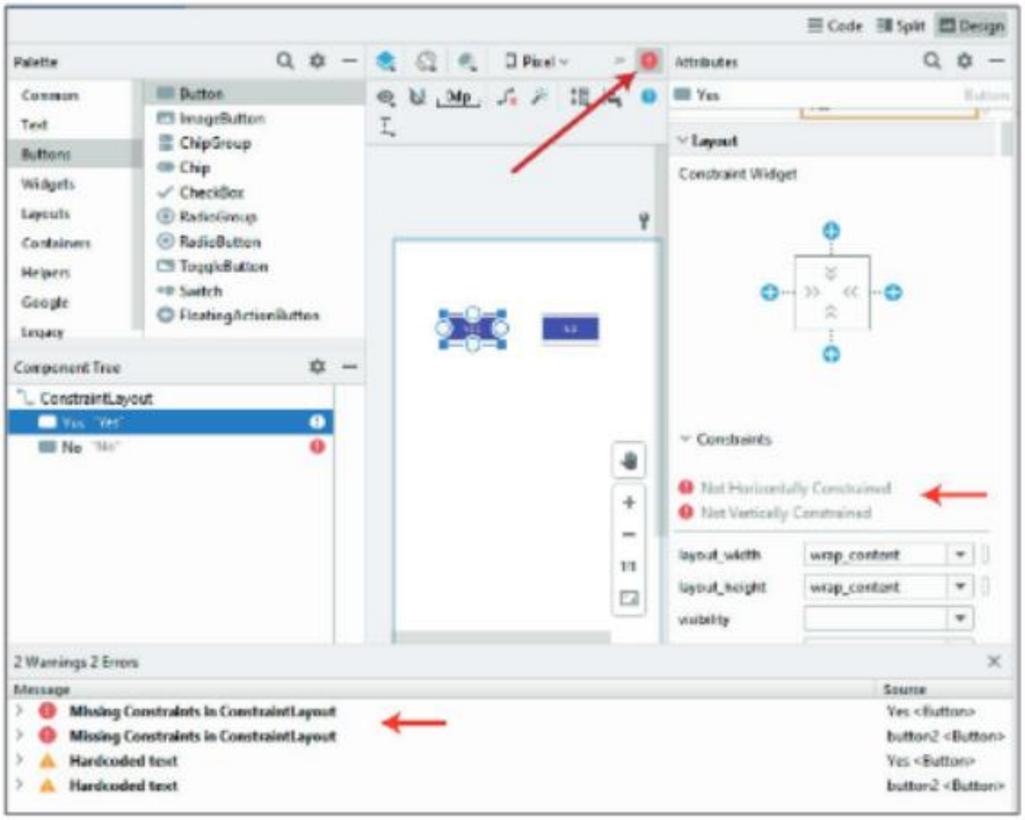
3- Change the attribute values of the **ID** and **Text** attributes as follows:

Button1	ID : Yes	Text: Yes
Button2	ID: No	Text: No

The following figure shows these two buttons:



As illustrated in the following figure, error messages will appear asking you to add constraints to these views (buttons) you added previously to this constraint layout activity. Also, if you click the **Problems** tab which exists above the status bar of the **Android Studio**, you will see these messages in details.



Not Horizontally Constrained
Not Vertically Constrained

layout_width	wrap_content
layout_height	wrap_content
visibility	

2 Warnings 2 Errors

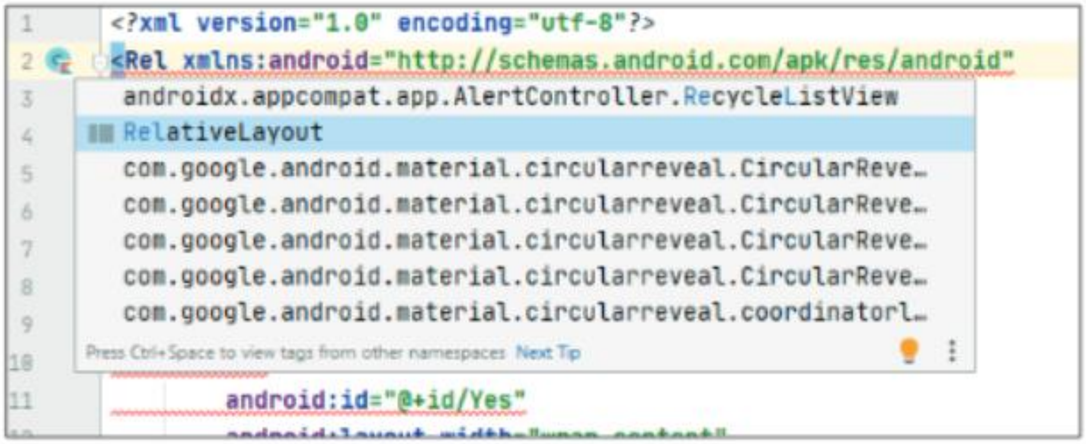
Message	Source
Missing Constraints in ConstraintLayout	Yes <Button>
Missing Constraints in ConstraintLayout	button2 <Button>
Hardcoded text	Yes <Button>
Hardcoded text	button2 <Button>

This is an important step because when you run your app, **Android Studio** has to know where each item should be located. Setting the constraints of these buttons should solve this issue; however, here we will test another technique of the activity layout which is the relative layout. By using this relative layout, there is no need here to set the constraints (margins) of each button, instead of that, configure the relative layout as it is explained in the next step:

4- Open **activity_main.xml** file in Code mode. You will get the following file:

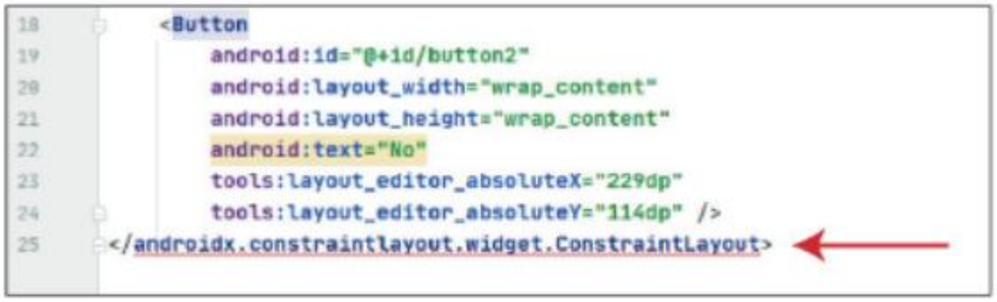
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/No"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/Yes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Yes"
        tools:layout_editor_absoluteX="77dp"
        tools:layout_editor_absoluteY="114dp" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="No"
        tools:layout_editor_absoluteX="229dp"
        tools:layout_editor_absoluteY="114dp" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

5- Replace `androidx.constraintlayout.widget.ConstraintLayout` with `RelativeLayout` as illustrated in the following figure:



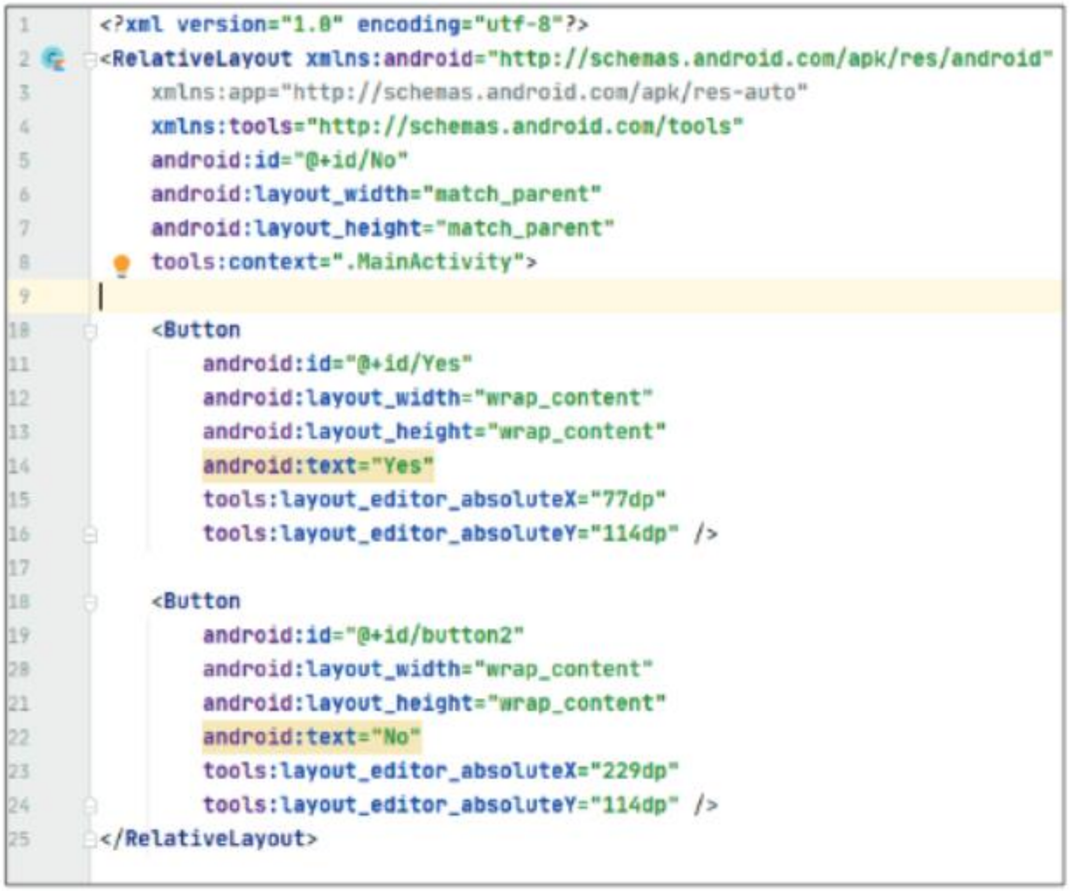
```
<?xml version="1.0" encoding="utf-8"?>
<Rel xmlns:android="http://schemas.android.com/apk/res/android"
    androidx.appcompat.app.AlertController.RecycleListView
    RelativeLayout
    com.google.android.material.circularreveal.CircularReve…
    com.google.android.material.circularreveal.CircularReve…
    com.google.android.material.circularreveal.CircularReve…
    com.google.android.material.circularreveal.CircularReve…
    com.google.android.material.circularreveal.coordinatorl…
    Press Ctrl+Space to view tags from other namespaces  Next Tip
        android:id="@+id/Yes"
```

6- Replace the last line of the XML code which is illustrated in the figure below with: **`</RelativeLayout>`**



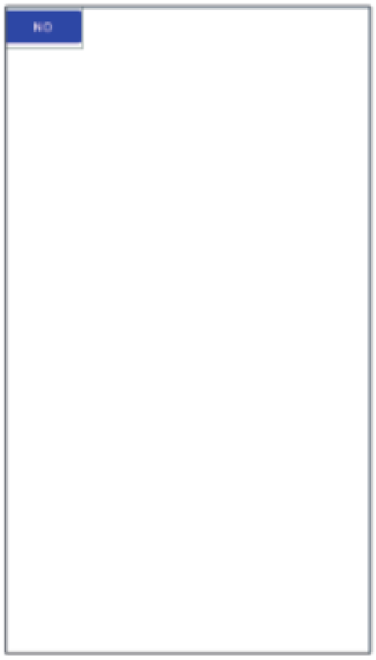
```
    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="No"
        tools:layout_editor_absoluteX="229dp"
        tools:layout_editor_absoluteY="114dp" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The XML file should be as follow:



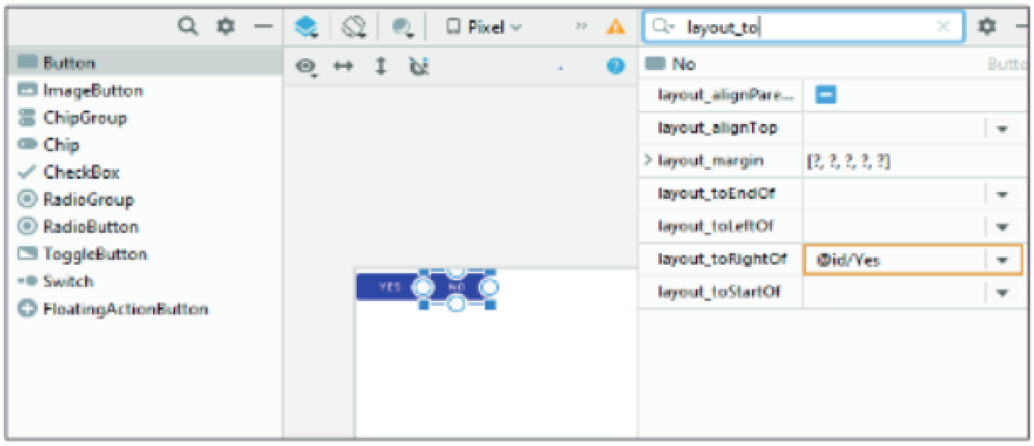
```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/No"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/Yes"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Yes"
        tools:layout_editor_absoluteX="77dp"
        tools:layout_editor_absoluteY="114dp" />

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="No"
        tools:layout_editor_absoluteX="229dp"
        tools:layout_editor_absoluteY="114dp" />
</RelativeLayout>
```

7- Open the **activity_main.xml** file in **Design** mode. The following figure displays the app interface in the design mode and the run result of this app. Here the two buttons moved to the point (0,0) because they don't have configured relative alignments to other widgets.



11:00
Lesson06
NO

8- Select the No button, open the **Attributes** panel, type **layout_to** in the search area as illustrated in the following figure, and change the attribute value of the attribute **layout_toRightOf** to: **@id/Yes** as illustrated in the following figure:

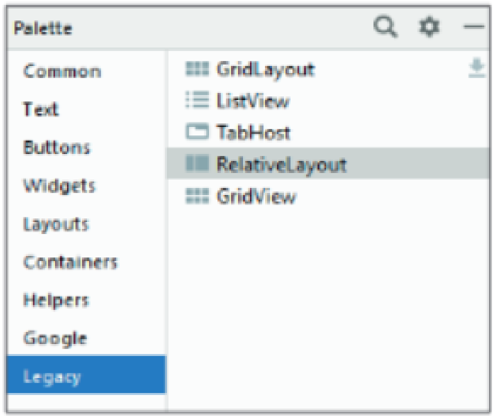


layout_alignPare...	
layout_alignTop	
layout_margin	[?, ?, ?, ?, ?]
layout_toEndOf	
layout_toLeftOf	
layout_toRightOf	@id/Yes
layout_toStartOf	

If you open the **activity_main.xml** file in the Code mode, you will find that the following attribute has been added to the **No** button tag: `android:layout_toRightOf="@id/Yes"`

Now, you may move the **No** button to the right side a little bit, but you can't make the **Yes** button on the right side of the **No** button.

You may add the **RelativeLayout** directly to your app interface from the Palette panel as illustrated in the following figure:

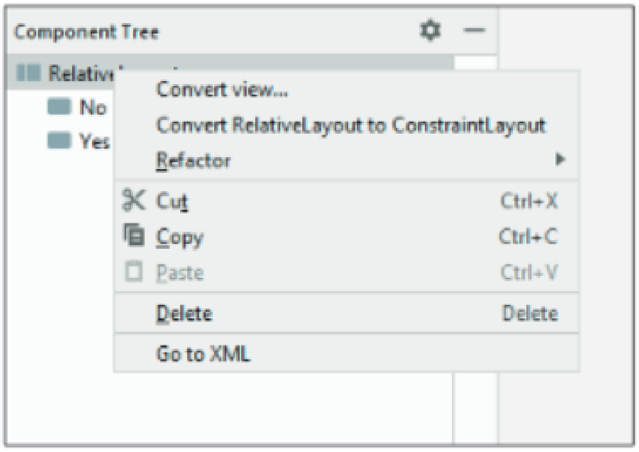


TableLayout

TableLayout arranges its children (views) into rows and columns. A **TableLayout** consists of a number of **TableRow** objects, each defining a row.

The following steps show how to create a table layout:

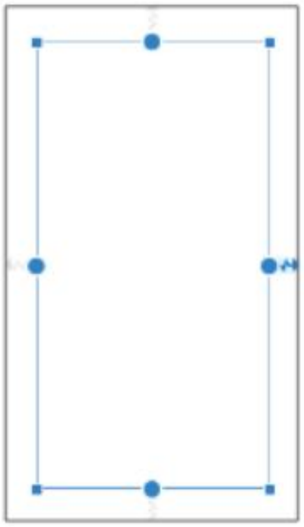
1- From the **Component Tree** panel of the **activity_main.xml** file, right click the **RelativeLayout** which you have created in the previous section as illustrated in the figure below, and select **Convert view.**



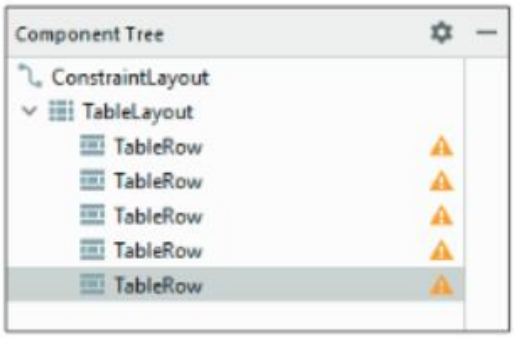
2- Select **ConstraintLayout**, and then click **Apply.**

3- From the **Components Tree,** delete the two existing buttons (Yes & No).

4- From the Palette panel (Layouts), drag a **TableLayout** view to the interface of the **activity_main.xml** file (in Design mode). Resize and set its constraints of this **TableLayout**, and you should get an interface similar to the following figure:



5- Now, you should use the **TableRow** view to divide this table to more than one row. Drag a **TableRow** view from the palette (Layouts) and drop it inside the **TableLayout** border which you have created in the previous step. You should get the following figure:



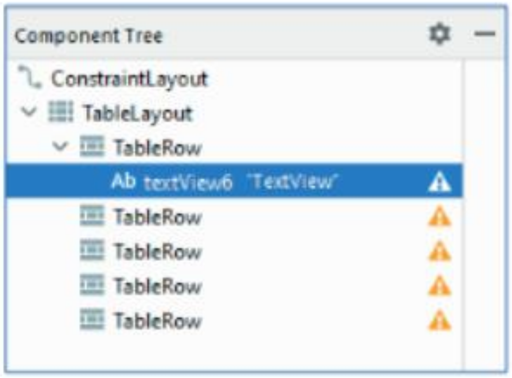
Component Tree

ConstraintLayout
- TableLayout
  - TableRow
  - TableRow
  - TableRow
  - TableRow
  - TableRow

6- Open the **activity_main.xml** file in the Code mode. As you see in the following code, the table consists of five **TableRow** (rows). You can add or remove the **TableRow** tags depending on the design you want. You can also add to each **TableRow** tag the widgets you need (like text, buttons and others). The table layout is similar to an HTML table, where the web designer creates and includes an HTML web page with text and images.

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

    <TableLayout
        android:layout_width="343dp"
        android:layout_height="660dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <TableRow
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

        <TableRow
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

        <TableRow
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

        <TableRow
            android:layout_width="match_parent"
            android:layout_height="match_parent" />

        <TableRow
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </TableLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

7- If you want to add a **TextView** to your specific **TableRow**, you can do that easily by dragging the **TextView** view from the **Palette** panel to the **TableRow** which you want in the **Component Tree** panel as you see in the following figure:



Or if you check this **TableRow** XML tag, you will find that it has a child **TextView** tag as you see in the following figure:

```
<TableLayout
    android:layout_width="343dp"
    android:layout_height="660dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <TextView
            android:id="@+id/textView6"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="TextView" />
    </TableRow>
```

Note: You can create a **TableLayout** inside or (sub of) Constraint or Relative layouts.

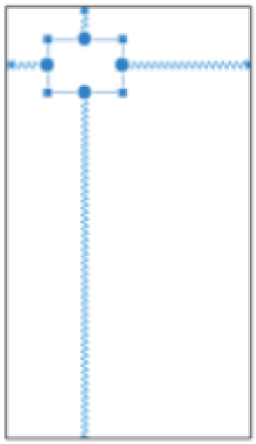
8- Delete the **TableLayout** view from the **Component Tree** panel to make this interface ready for the next practice.

Frame Layout

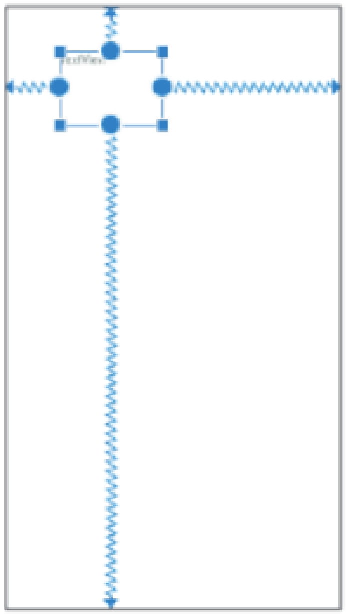
FrameLayout is designed to block out an area on the screen to display a single item. Generally, **FrameLayout** should be used to hold a single child view because it can be difficult to organize child views in a way that would scalable to different screen sizes without the children overlapping each other.

Using the same **activity_main.xml** file in the Design mode, follow the following steps:

1- Drag the **FrameLayout** view from the Palette panel to your app interface (**activity_main.xml** in **Design** mode), reside the layout, and set its constraints as illustrated in the following figure:



2- Drag a **TextView** inside this **FrameLayout** as illustrated in the following figure:



Note, that you have full control to move over **FrameLayout,** so this **FrameLayout** works like an area of the interface which gives us a full control to move the child item (**TextView**) as we want. This makes the control on the location of each interface item easier.

Android Styles and Themes

Building your Android app requires more than just writing a Kotlin code. As you have already seen in this course, you need to build the layout of your activities to provide the app user interface. These layout files are XML files saved under **/res/layout,** and they are just one example of the many resource files you may add to your Android app.

If you need a static and a contact data (images, strings… etc.), you should add them as resources in your app i.e. externalize them. Adding your app resources in separate folders improves your development experience and yields a more robust app for several reasons:

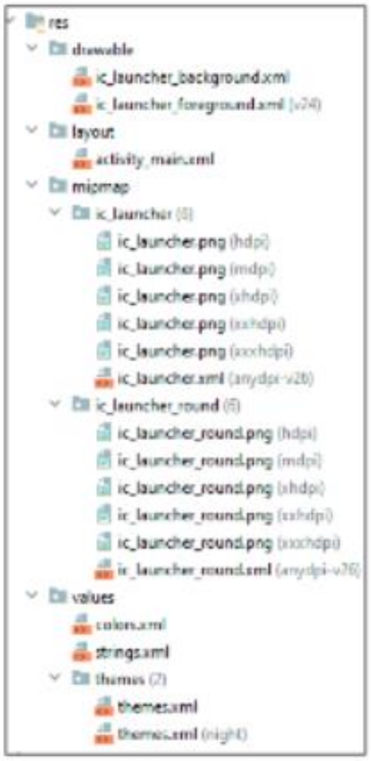
1) Maintaining your application will be much easier since your business code is separate from the static interface code.

2) Easier debugging of the app because resources are independent from the code.

3) Reaching a much wider user base and covering much more Android-powered devices by qualifying your resources; i.e. providing multiple instances of the same resource for different device configurations. This is essential to tackle the Android fragmentation issue.

4) Improving the user interface and user experience.

5) Easier development since you do not need to build every resource programmatically in your code.



```
res
  drawable
    ic_launcher_background.xml
    ic_launcher_foreground.xml (v24)
  layout
    activity_main.xml
  mipmap
    ic_launcher (6)
      ic_launcher.png (hdpi)
      ic_launcher.png (mdpi)
      ic_launcher.png (xhdpi)
      ic_launcher.png (xxhdpi)
      ic_launcher.png (xxxhdpi)
      ic_launcher.xml (anydpi-v26)
    ic_launcher_round (6)
      ic_launcher_round.png (hdpi)
      ic_launcher_round.png (mdpi)
      ic_launcher_round.png (xhdpi)
      ic_launcher_round.png (xxhdpi)
      ic_launcher_round.png (xxxhdpi)
      ic_launcher_round.xml (anydpi-v26)
  values
    colors.xml
    strings.xml
    themes (2)
      themes.xml
      themes.xml (night)
```

Android SDK allows developers to create the user interface theme of their applications using XML resource files. Defining the style of user interfaces involves specifying values for colors, fonts, dimensions, buttons…etc. When a resource is used to define the theme of a certain view, it is called a **style**. However, when the same style is applied on the whole activity or app, it is called a **theme**.

Styles and themes on Android development allow you to separate the details of your app design from the UI (User Interface) structure and behavior, similar to Cascading Style Sheets (CSS) in web design.

A **style** is a collection of attributes that specify the appearance for a single View. A style can specify attributes such as font color, font size, background color, and much more.

A **theme** is a collection of attributes that's applied to an entire app, activity, or view hierarchy—not just an individual view. You can say that the app theme consists of collection of many styles.

Like any other resources, the themes are defined and saved in XML files under **/res** folder – specifically under **/res/values/themes/**. When you create a new project using Android Studio, a default themes resource file called **themes.xml** is created under **/res/values/ themes/**.

Example:

1-From Android Studio, click **File → New → New Project.**

2- Select **Empty Activity**, and click **Next.**

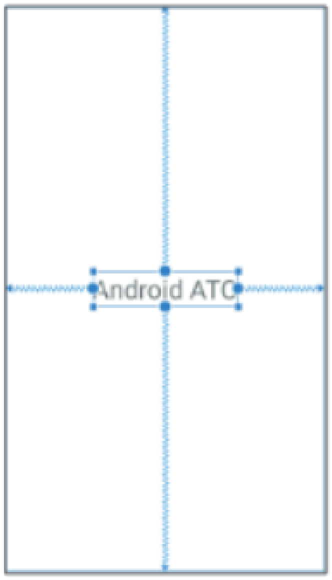
3- Enter the application Name: **Lesson06_Styles and** then click **Finish**.

4- To learn how you can use styles in Android app, open **activity_main.xml** in **Design** mode, select the

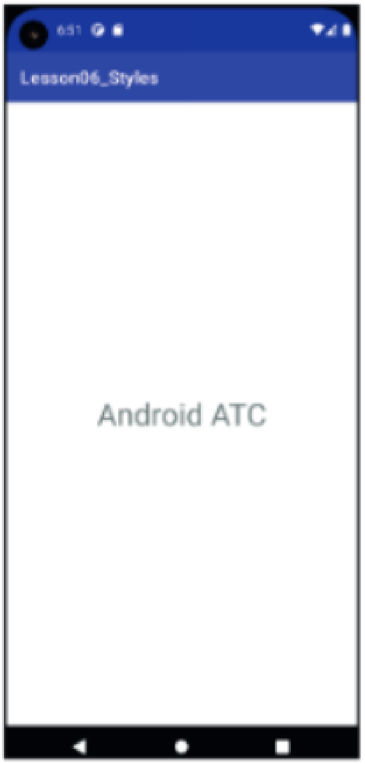
TextView: "Hello World!", click the **Attributes** tab, and change the following attributes' values as illustrated in the following table:

text	Android ATC
textSize	34sp

The layout interface should have the following figure:

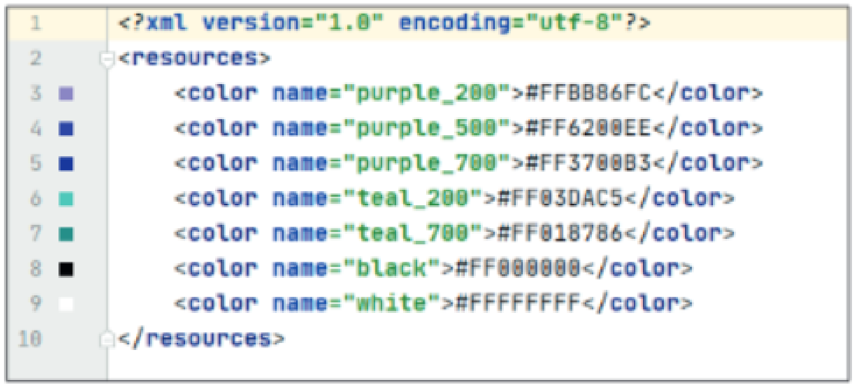


5- Run your app, you should have the following rub result:



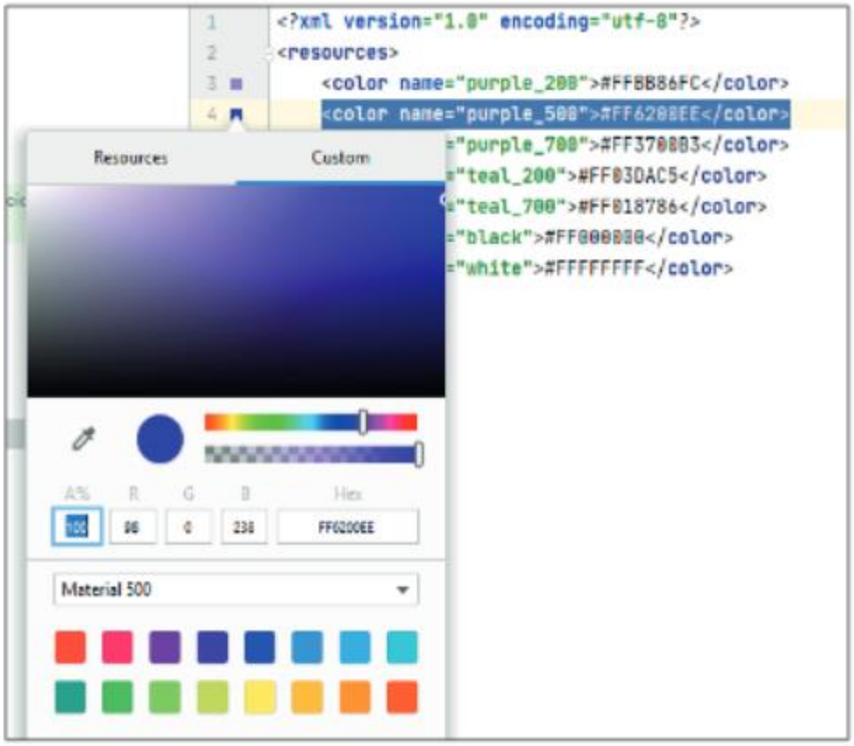
6- You can modify your default app title or status bars' colors if you change the colors attributes values which are in the **colors.xml** file (**app → res → values → colors.xml**).

Note: The Android status bar is where you will find the Wi-Fi, Bluetooth, mobile network, battery, time, alarm, etc.



```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
</resources>
```

7- For example, to change the color of your app title to red color, **click** the blue (square) for the color name: **purple_500**, and then select **Red** color as shown in the figure below:

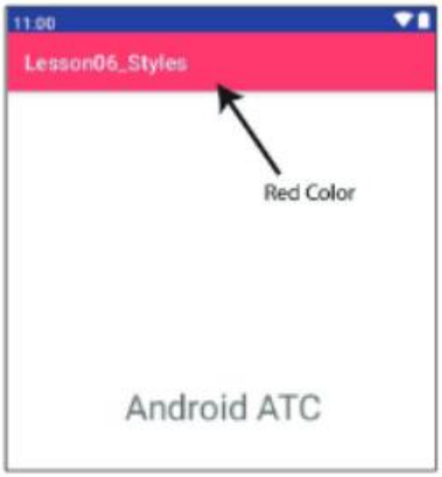


```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#FF6200EE</color>
                  ="purple_700">#FF3700B3</color>
                  ="teal_200">#FF03DAC5</color>
                  ="teal_700">#FF018786</color>
                  ="black">#FF000000</color>
                  ="white">#FFFFFFFF</color>
```

A% 100 R 98 G 0 B 238 Hex FF6200EE

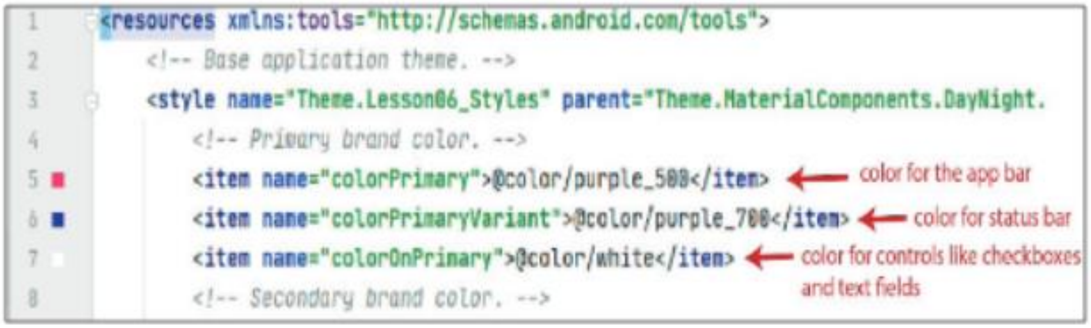
Material 500

8- The run result of your app should be as follows:



Here, you should ask yourself the following question: Which part of this app has the configuration of app title color or declare that the app title bar color name is: purple_500?

The answer of this question is the **themes.xml** file (**app** → **res** → **values** → **themes** → **themes.xml**).



```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Lesson06_Styles" parent="Theme.MaterialComponents.DayNight.
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <!-- Secondary brand color. -->
```

This **themes.xml** file is considered the main reference for all your app interfaces. Also, you may change the colors in this **themes.xml** file directly without the need to edit them in **colors.xml** file.

The main idea of configuring all the styles of this app in this **themes.xml** file is to be sure that these configurations will apply for all your app activities. You will find similarity between using styles and themes in Android and HTML style tags and CSS. Using themes.xml file is more like using CSS in HTML and using style tag in Android is similar to using inline style tag or style attribute in HTML language.

9- If you want to change the app activities background color to yellow color, you should configure this color first in the colors.xml file (**app → res → values → colors.xml**). As you see in the following code of **colors.xml** file, I gave this color a name called **YellowBackground** as illustrated in the gray highlighted color in the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="purple_200">#FFBB86FC</color>
    <color name="purple_500">#E91E63</color>
    <color name="purple_700">#FF3700B3</color>
    <color name="teal_200">#FF03DAC5</color>
    <color name="teal_700">#FF018786</color>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
    <color name="YellowBackground">#FFEB3B</color>
</resources>
```

10- Open the **themes.xml** file, then add the gray highlighted code as illustrated in the following code:

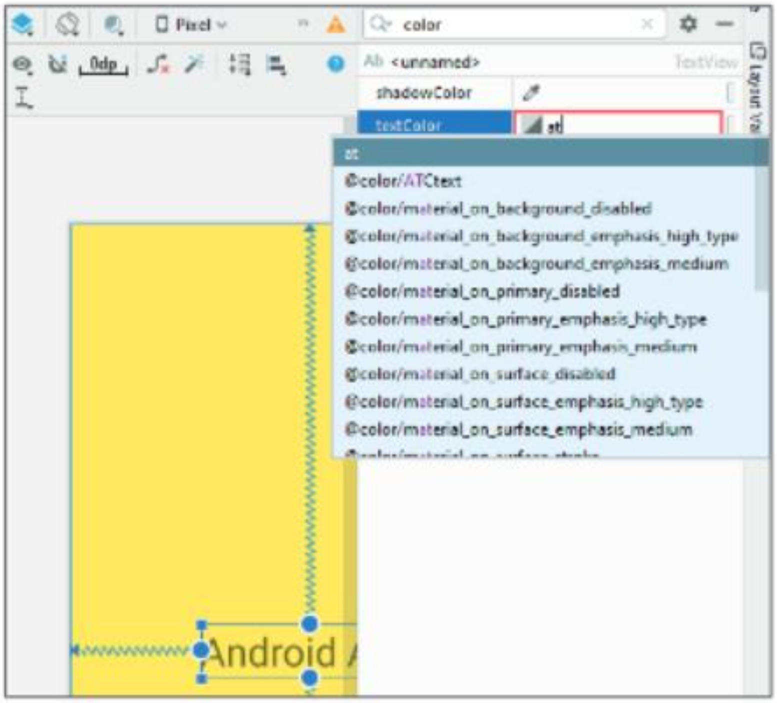
```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Lesson06_Styles" parent="Theme.
MaterialComponents.DayNight.DarkActionBar">
      <!-- Primary brand color. -->
      <item name="colorPrimary">@color/purple_500</item>
      <item name="colorPrimaryVariant">@color/purple_700</item
```

```
<item name="colorOnPrimary">@color/white</item>
    <item name="android:windowBackground">@color/YellowBackground</item>
    <!-- Secondary brand color. -->
    <item name="colorSecondary">@color/teal_200</item>
     <item name="colorSecondaryVariant">@color/teal_700</item>
     <item name="colorOnSecondary">@color/black</item>
     <!-- Status bar color. -->
     <item name="android:statusBarColor" tools:targetApi="l">?attr/
colorPrimaryVariant</item>
     <!-- Customize your theme here. -->
    </style>
</resources>
```

11- If you want to add a new font color to use it later with a **TextView** view, you should add this color to the **colors.xml** file, then add it to the **TextView** view which you want. Here, in this example, open the **colors.xml** file, and add a new color attribute with a value green color as follows:

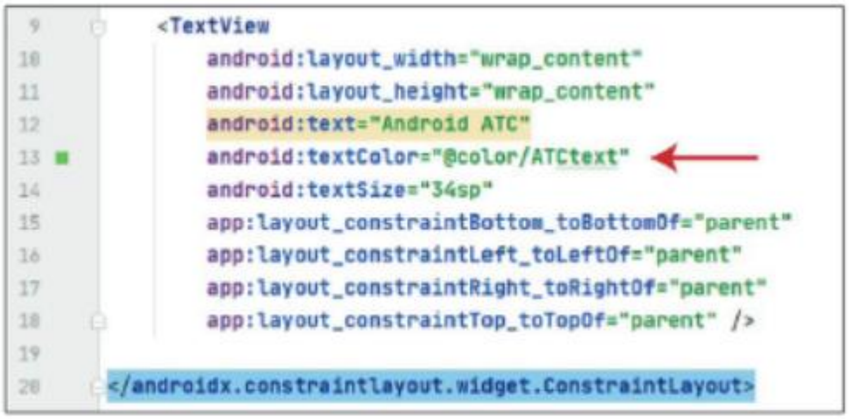
```
<color name="ATCtext">#00ff00</color>
```

12- Open the **activity_main.xml** file in Design mode, select the **TextView**: **Android ATC,** click the **Attributes** tab, in the search area type **color** and press **Enter**. In the **textColor** attribute value text filed, type the color name: **ATCtext**. As illustrated in the figure below, you should get this color name with the list. Select this font color name, and then press **Enter**. The font color of the **TextView** (Android ATC) should be changed to green color.



@color/ATCtext
@color/material_on_background_disabled
@color/material_on_background_emphasis_high_type
@color/material_on_background_emphasis_medium
@color/material_on_primary_disabled
@color/material_on_primary_emphasis_high_type
@color/material_on_primary_emphasis_medium
@color/material_on_surface_disabled
@color/material_on_surface_emphasis_high_type
@color/material_on_surface_emphasis_medium

Also, you can change this font color if you add the below attribute to this **TextView** (Android ATC) in this XML file ,but in the **Code** mode, as illustrated in the following figure:



```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Android ATC"
        android:textColor="@color/ATCtext"
        android:textSize="34sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

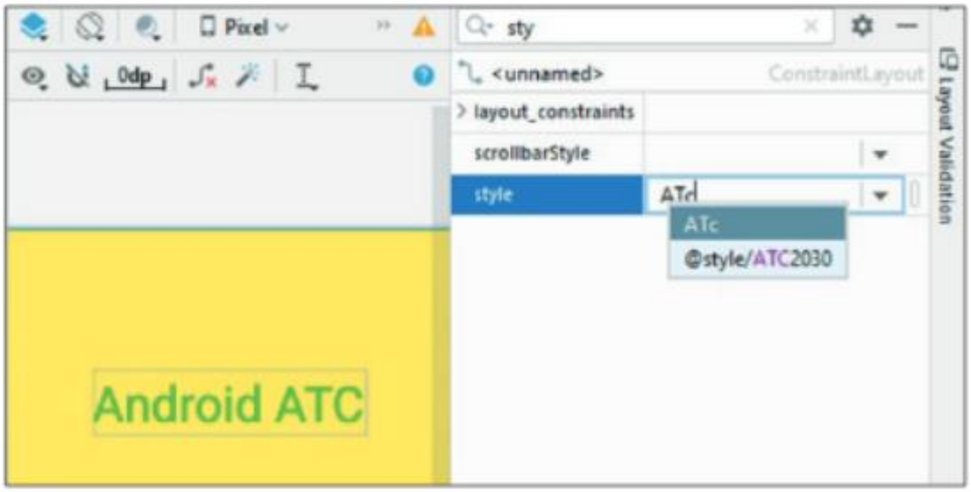
Note: To summarize, the **colors.xml** file includes three default colors which produce the default color of your app titles and widgets. You can also add other colors to be used later while formatting your app layout contents. But why do you need to add the new colors in the **colors.xml** file if you can give each text the color you want directly by using the "**textColor**" attribute? Adding the new color in the **colors.xml** file helps the app designer to use the same colors for titles or headers in the app to ensure consistency.

13- If you want to make your text widgets in your entire app have the same format such as font color, font size and font family type, you should create a **style** in the **themes.xml** file that has all these font format specifications of font type, font size, color, and then apply this style name on the **TextView** which you want to have this style or format. To do that, open **themes.xml** file and add a new style as illustrated in the gray highlighted code below:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Lesson06_Styles" parent="Theme.
MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <item name="android:windowBackground">@color/
YellowBackground</item>
        <!-- Secondary brand color. -->
        <item name="colorSecondary">@color/teal_200</item>
        <item name="colorSecondaryVariant">@color/teal_700</item>
        <item name="colorOnSecondary">@color/black</item>
        <!-- Status bar color. -->
        <item name="android:statusBarColor"
tools:targetApi="l">?attr/colorPrimaryVariant</item>
```

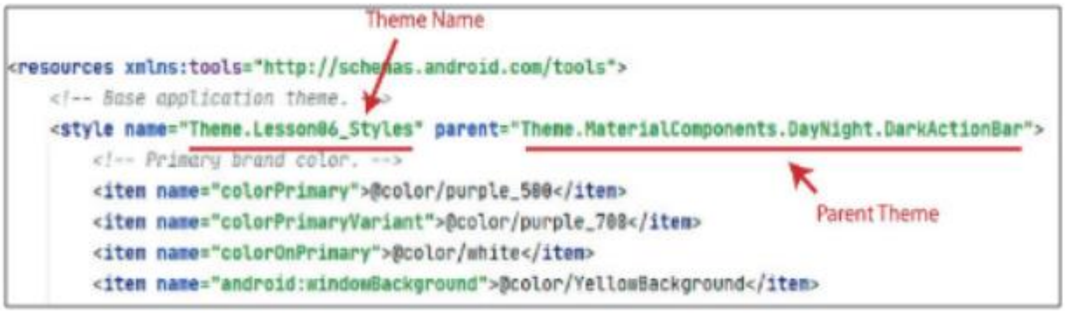
```
        <!-- Customize your theme here. -->
    </style>

    <style name="ATC2030">
        <item name="android:textSize">16sp</item>
        <item name="android:textColor">#00ff00</item>
        <item name="android:fontFamily">cursive</item>
    </style>

</resources>
```

14- Open the **activity_main.xml** in **Design** mode, select the TextView: **Android ATC**, click the **Attributes** tab, type: "**style**" in the search area, then type the style attribute value: **ATC2030** as illustrated in the following figure, and press **Enter**.



Directly you should find that all the new font size, color, and type have been applied on this **TextView** (Android ATC).

14- Your app or any new app you will create in future, using Android Studio, will have a default theme. This theme will apply on all your app activities. For example, if you open this app theme file: **themes.xml**, you will find the default theme name is: **Theme.Lesson06_Styles** as illustrated in the following figure. This theme does not cover all other activity styles such as banners colors, top title color, and others. Therefore, you should inherit the other theme styles from other existing themes (parent theme) like: **Theme.MaterialComponents.DayNight. DarkActionBa.**



```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Theme.Lesson06_Styles" parent="Theme.MaterialComponents.DayNight.DarkActionBar">
        <!-- Primary brand color. -->
        <item name="colorPrimary">@color/purple_500</item>
        <item name="colorPrimaryVariant">@color/purple_700</item>
        <item name="colorOnPrimary">@color/white</item>
        <item name="android:windowBackground">@color/YellowBackground</item>
```

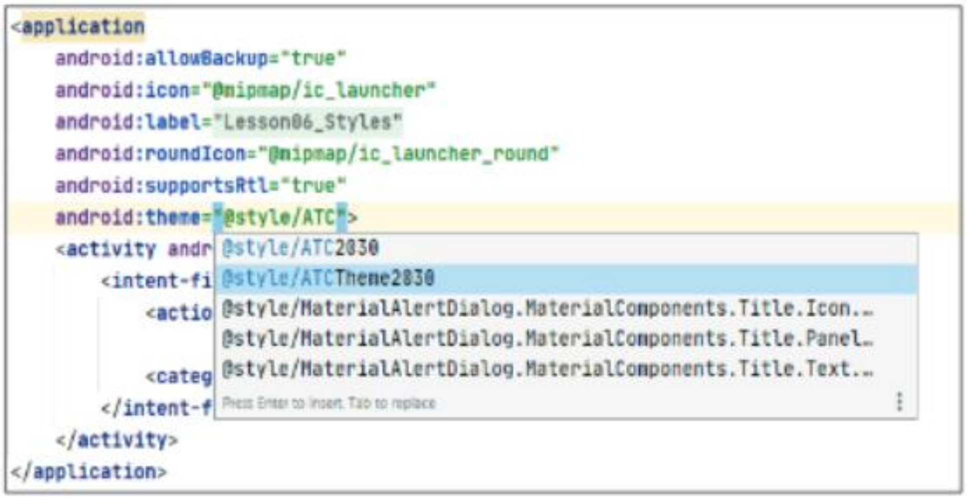
You can create your own theme in this **themes.xml** file, and add to it all the styles which you want, but this depends on the style name you created before in styles.xml file. For example, add at the end of the **themes.xml** file and before the `</resources>` and tag the following code which is related to a new theme which is called **ATCTheme2030,** and inherit its styles from another parent theme which is called: Theme.AppCompat.DayNight

```
<style name="ATCTheme2030" parent="Theme.AppCompat.DayNight">
    <item name="android:textSize">14sp</item>
    <item name="android:textColor">#00ff00</item>
    <item name="android:fontFamily">cursive</item>

</style>
```

15- Now, the **themes.xml** file includes two themes. The question is where you can configure your app to set its theme. You can do that by configuring: **AndroidManifest.xml** (app → Manifest → AndroidManifest.xml) file.

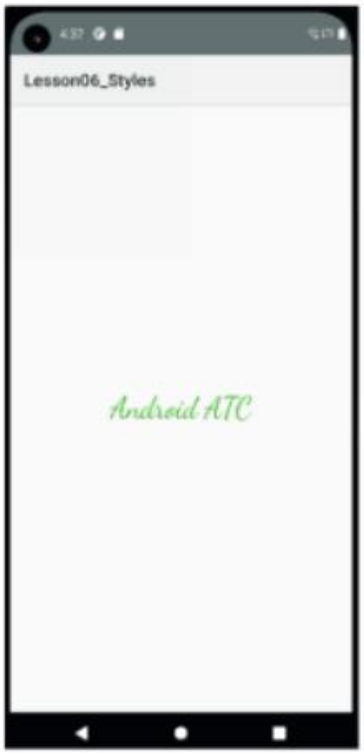
Open **AndroidManifest.xml** file, and you will find the name of the applied theme as value of the attribute: **android:theme**

Replace the **android:theme** attribute value : **@style/Theme.Lesson06_Styles** with **@style/ATCTheme2030** as illustrated in the following figure:



```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="Lesson06_Styles"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/ATC">
    <activity andr @style/ATC2030
        <intent-fi @style/ATCTheme2030
            <actio @style/MaterialAlertDialog.MaterialComponents.Title.Icon..
                   @style/MaterialAlertDialog.MaterialComponents.Title.Panel..
            <categ @style/MaterialAlertDialog.MaterialComponents.Title.Text..
        </intent-f Press Enter to insert, Tab to replace
    </activity>
</application>
```

16- Stop, then run your app. You will find that the new theme has been applied on your app as illustrated in the following figure:



App Manifest

Every application must have namely an **AndroidManifest.xml** file in its root directory (app → Manifest → AndroidManifest.xml). The manifest file provides the Android system with essential information about your app which the system needs before it can run any of the app's codes.

You have already used this file in the previous example to apply a new theme to your activity using `android:theme="@style/ ATCTheme2030"` attribute of the application tag. This file has an important role in the app security settings. In this course, within the next lessons, you will get more details with examples about the importance of changing the settings of this file to your app.

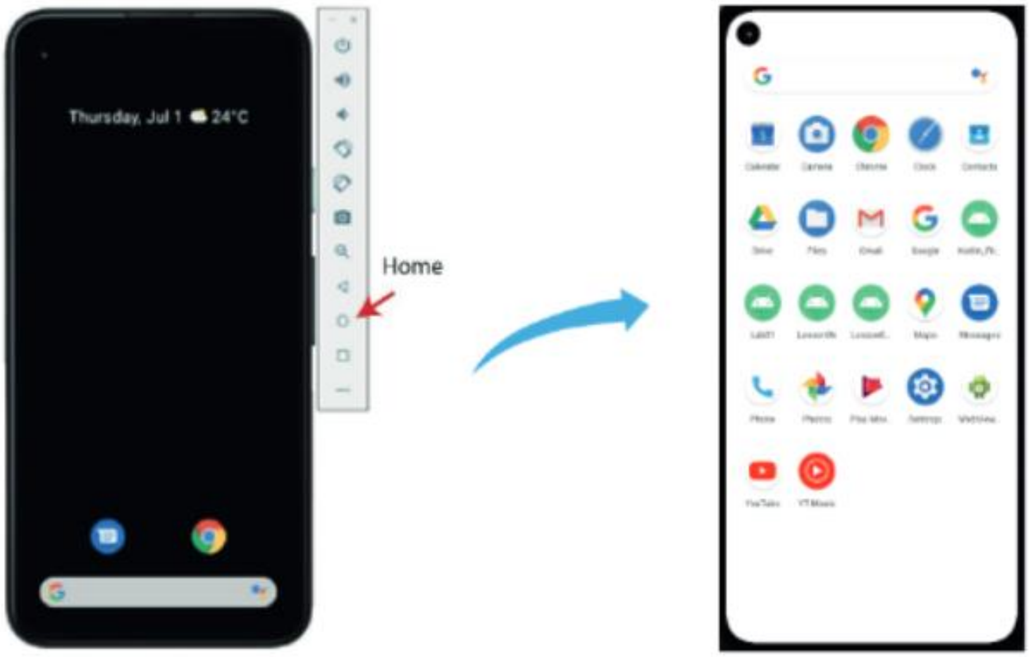
17- You can change your app name if you change it in the **strings.xml** file (app → res → values).
Open **strings.xml** file, as illustrated in the code below, replace the app name: **Lesson06_Styles** with **TestApp**, then run your app and check the changes.

```
<resources>
    <string name="app_name">Lesson06_Styles</string>
</resources>
```

App Icons

Android Studio includes a tool called **Image Asset Studio** that helps you generate your own app icons from material icons, custom images, and text strings.

As illustrated in the figure below, if you click the **Home** button of your emulator tool bar, then swipe up the emulator screen, you should get a list of your apps icons which you have in your phone emulator. As you see in the figure below, there is an app icon for lesson06 and for your current app (lesson06_styles). You will find that your Android app icon is the Android logo. This is the default icon setting for any Android app. Click on your app icon to startup your app again.



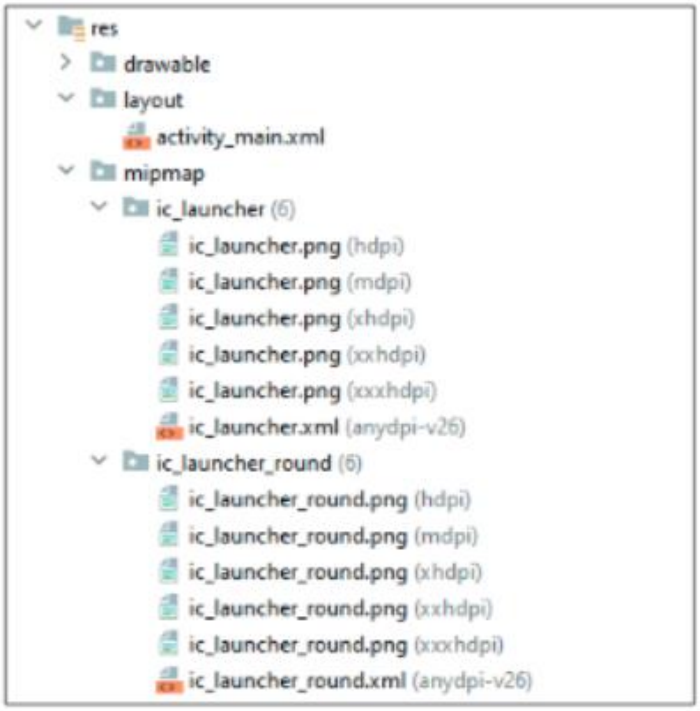
In this part of the lesson, you will configure your app icons for Android devices. Also, this icon will represent your app on your customers' smart devices, and in Google Play Stores.

You can import your own images and adjust them for the icon type. Image Asset Studio supports the following file types: **PNG** (preferred), **JPG** (acceptable), and **GIF** (discouraged).

Also, Image Asset Studio lets you type a text string in a variety of fonts and places it on an icon. It converts the text-based icon into PNG files for different densities. You can use the fonts that are installed on your computer.

You may ask a graphic designer to create your app icon, use Adobe Illustrator software, search on Google images, or use a free logo design software web site. The App icon is an image which has an extension JPEG or PNG. It is better to use PNG image format because PNG supports transparency feature while JPEG does not.

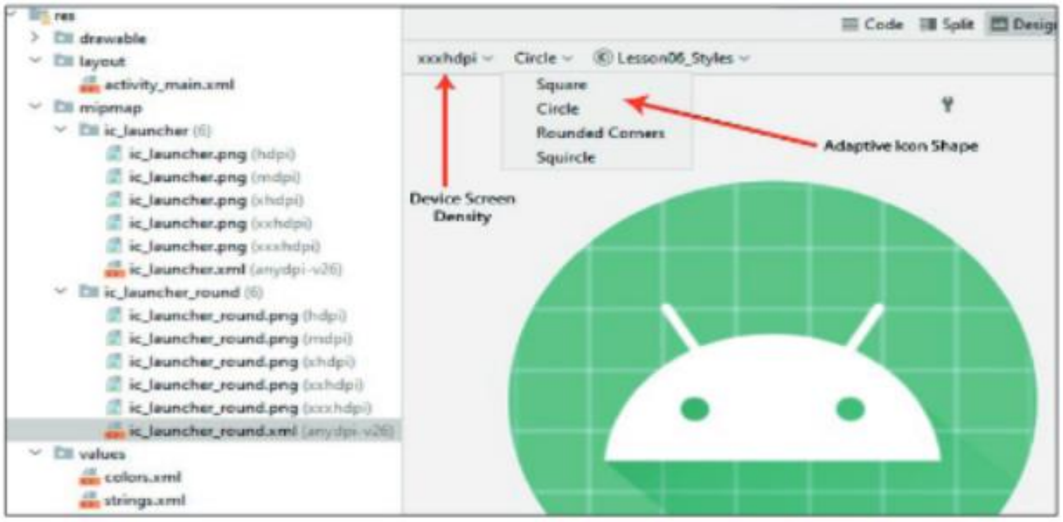
When you create your app using Android Studio, your app icon by default will be Android logo. The **mipmap** folder (app → res → mipmap). This folder as illustrated in the following figure includes two sub folders, **ic_launcher** and **ic_launcher_round.**



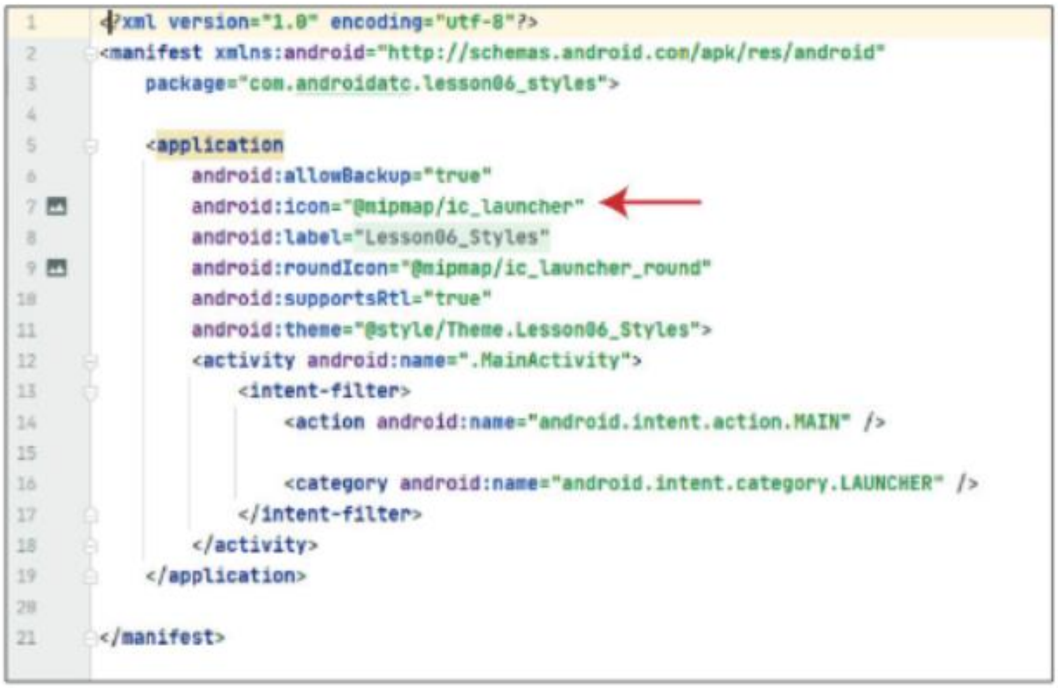
```
res
  drawable
  layout
    activity_main.xml
  mipmap
    ic_launcher (6)
      ic_launcher.png (hdpi)
      ic_launcher.png (mdpi)
      ic_launcher.png (xhdpi)
      ic_launcher.png (xxhdpi)
      ic_launcher.png (xxxhdpi)
      ic_launcher.xml (anydpi-v26)
    ic_launcher_round (6)
      ic_launcher_round.png (hdpi)
      ic_launcher_round.png (mdpi)
      ic_launcher_round.png (xhdpi)
      ic_launcher_round.png (xxhdpi)
      ic_launcher_round.png (xxxhdpi)
      ic_launcher_round.xml (anydpi-v26)
```

Mipmap folder is the default location for all your app icons with different densities. Because your app will run on different types of smart Android devices, and each device has its own screen size and resolution; therefore, you need to have these icons in different resolutions.

ic_launcher folder includes these icons in a square shape, while the ic_launcher_round includes the app icons in a circular shape.

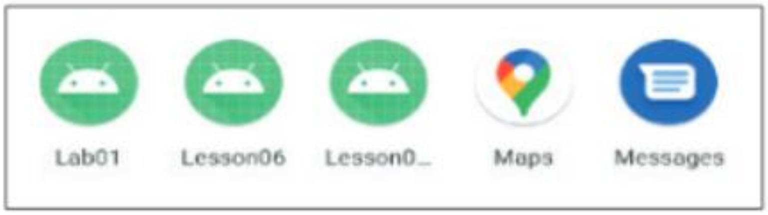


Your app already has been configured to use the icons which are existing in the **ic_launcher** folder. |This configuration is exiting in the **AndroidManifest.xml** file of any Android app. Open your **AndroidManifest.xml,** and you will find the following attribute:



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidatc.lesson06_styles">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Lesson06_Styles"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Lesson06_Styles">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

As you saw in the previous step, your app icon at your phone emulator has a circular shape with the Android icon, as illustrated in the following figure:

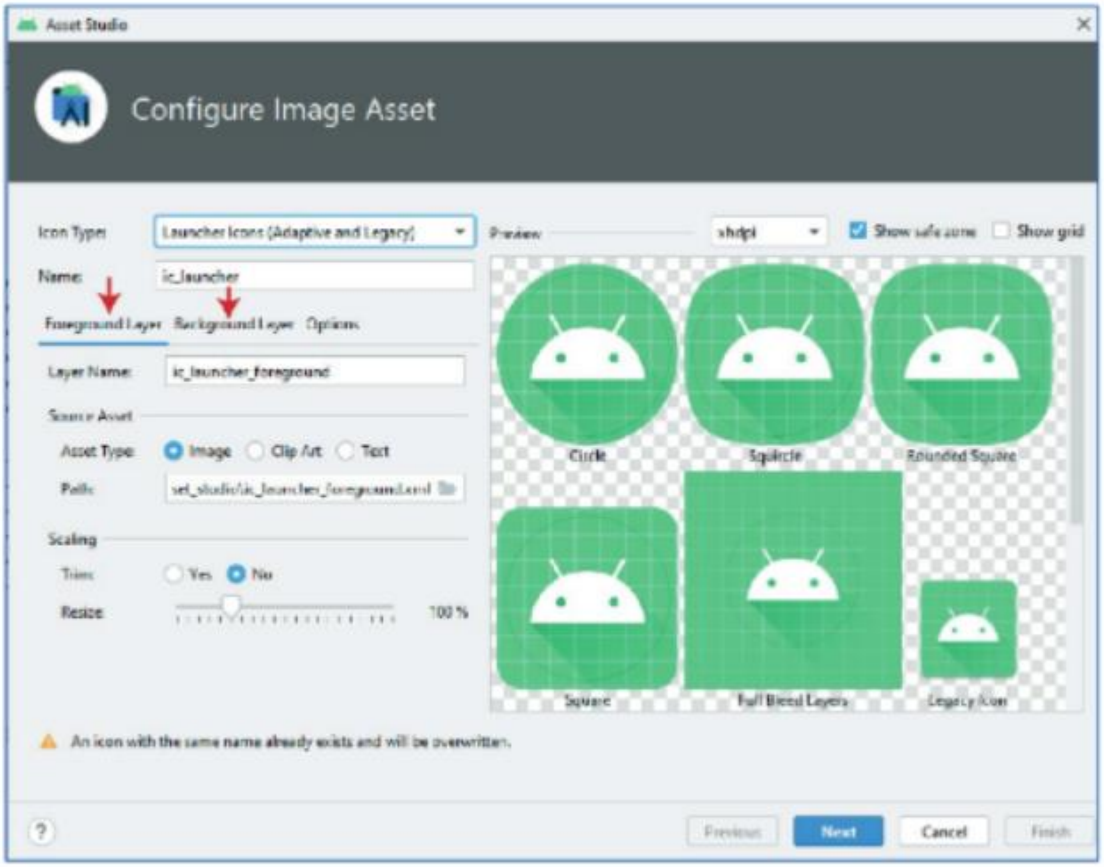


Your app adaptive launcher consists of two layers, a background and a foreground layers.

To change your default app icons, you should use the Asset Studio in Android Studio. To do that, follow the following steps:

1- Right click **app → New → Image Asset**

You will get the following dialog box:



Icon Type: Launcher Icons (Adaptive and Legacy)

Name: ic_launcher

Layer Name: ic_launcher_foreground

Source Asset

Asset Type: [✓ Image] Clip Art Text

Path: set_studio/ic_launcher_foreground.xml

Scaling

Trim: Yes [✓ No]

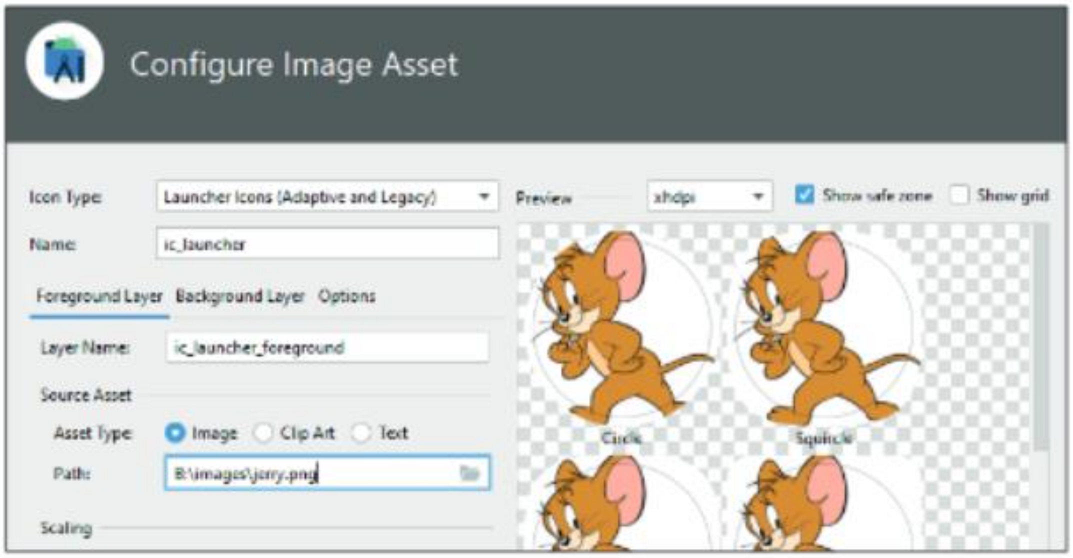
Resize: 100 %

xhdpi [✓ Show safe zone] Show grid

An icon with the same name already exists and will be overwritten.

Your app icon consists of two layers. In the above dialog box, you can configure one choice of the three types of assets (an image, a clip art and a text) for the "**Foreground Layer**" of your app icon. But you can select only an image or a color for the "**Background Layer**".

2- You can add any image (png or jpeg) as a foreground image by clicking the browse button to select your image path. In this example, we will use a Jerry carton image (from Tom and Jerry) as a **Foreground Layer** as illustrated in the following figure:



Configure Image Asset

Icon Type: Launcher Icons (Adaptive and Legacy)

Name: ic_launcher

Foreground Layer | Background Layer | Options

Layer Name: ic_launcher_foreground

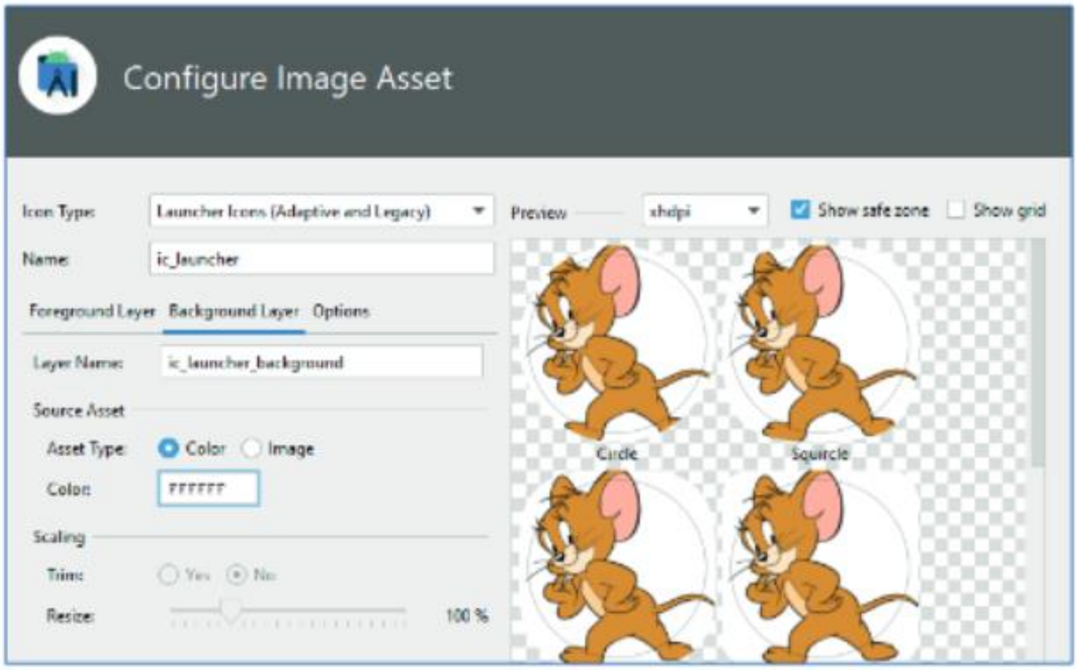
Source Asset

Asset Type: [✓ Image] Clip Art Text

Path: B:\images\jerry.png

Scaling

3- Select white color (FFFFFF) for the **Background Layer** as illustrated in the following figure:



Configure Image Asset

Icon Type: Launcher Icons (Adaptive and Legacy)

Name: ic_launcher

Foreground Layer | Background Layer | Options

Layer Name: ic_launcher_background

Source Asset

Asset Type: [✓ Color] Image

Color: FFFFFF

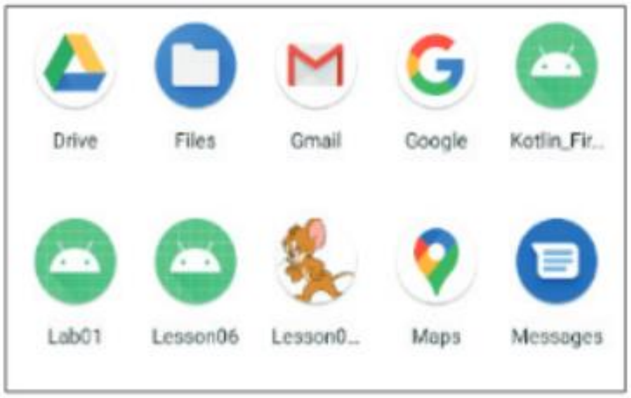
Scaling

Trim: Yes [✓ No]

Resize: 100 %

4- Click **Next**, and then click **Finish.**

5- Stop, then run your app. Check your app icon on your phone emulator. You should find your app icon as a Jerry image with white background as illustrated in the following figure:



Note: When a launcher requests an app icon, the framework returns to either `android:icon` or `android:roundIcon`, and this depends on the device build configuration. That's why apps should make sure to define both `android:icon` and `android:roundIcon` resources when responding to launcher intents.