UNIT:4 ANDROID USER INTERFACE DESIGN ESSENTIALS

- **Organizing User Interface using View Group and View: -**

Generally, every application is a combination of View and View Group.

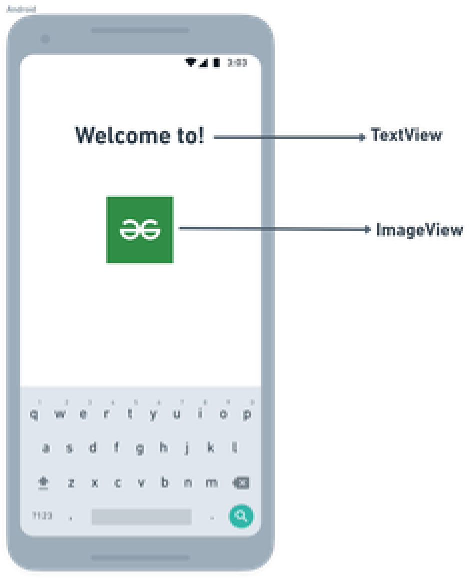
An android application contains a large number of activities and we can say each activity is one page of the application.

So, each activity contains multiple user interface components and those components are the instances of the View and View Group.

All the elements in a layout are built using a hierarchy of **View** and **View Group** objects.

View

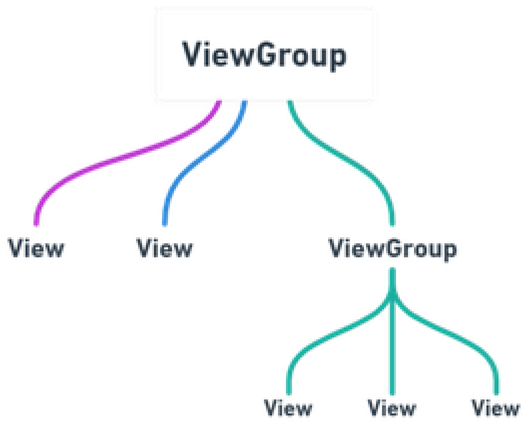
A **View** is defined as the user interface which is used to create interactive UI components such as Text View, Image View, Edit Text, Radio Button, etc., and is responsible for event handling and drawing. They are Generally Called Widgets.



View

VIEW GROUP:

A **View Group** act as a base class for layouts and layouts parameters that hold other Views or View Groups and to define the layout properties. They are Generally Called layouts.



View Group

- <u>**How To Create Layouts Using XML Resources: -**</u>

Creating layouts using XML resources is a fundamental aspect of Android app development. XML (Extensible Markup Language) is used to define the structure and appearance of user interfaces in Android. Here's a step-by-step guide on how to create layouts using XML resources:

1. **Create or Open a Project:**
   - Open Android Studio and create a new project or open an existing one.
2. **Navigate to the `res` Folder:**
   - In the **res** (resources) folder of your Android project, you'll find a folder named **layout**. This is where you store your XML layout files.
3. **Create a New XML Layout File:**
   - Right-click on the **layout** folder.
   - Choose **New -> Layout resource file**.
   - Provide a name for your layout file (e.g., **activity_main.xml**) and click **OK**.
4. **Design the Layout Using XML:**
   - Open the newly created XML file (**res/layout/activity_main.xml**) in the editor.
   - Use XML tags to define the structure and appearance of your layout. Common layout elements include **Linear Layout**, **Relative Layout**, **Constraint Layout**, etc.

5. **Preview the Layout:**
    - You can preview the layout by switching to the "Design" tab in Android Studio, or by using the "Preview" option.
6. **Adjust Attributes and Properties:**
    - Fine-tune your layout by adjusting attributes and properties of the XML elements. You can set attributes such as **layout_width**, **layout_height**, **text**, **id**, etc.
7. **Resource IDs and Referencing:**
    - Assign IDs to views using the **android:id** attribute. This allows you to reference these views in your Java/Kotlin code.
8. **Repeat for Additional Layouts:**
    - If your app has multiple screens, repeat the process to create XML layout files for each screen.

Once you've defined your layouts, you can use them in your activities by inflating them using **setContentView** in your **on Create** method.

Example: activity_main.xml file:

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

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

</ConstraintLayout>
```

Java File:

- `setContentView(R.layout.activity_main);`
- **How to Create Layouts Programmatically: -**

Creating layouts programmatically in Android involves instantiating **View** and **View Group** objects in code and configuring their properties. Here's a step-by-step guide on how to create layouts programmatically:

1. **Create a New Project:**
   - Open Android Studio and create a new project or open an existing one.
2. **Open the Activity File:**
   - Open the Java (or Kotlin) file corresponding to the activity where you want to create the layout programmatically.
3. **Import Necessary Packages:**
   - Import necessary packages for **View**, **View Group**, and other components you'll use in your code.
   - Java File:

```
import android.os.Bundle;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.LinearLayout;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
```

4. **Override on Create Method:**
   - Inside your activity class, override the **on Create** method.
5. **Create a View Group (e.g., Linear Layout):**
   - Create an instance of a **View Group** (e.g., **LinearLayout**, **RelativeLayout**, **ConstraintLayout**).
   - Java File:

```
LinearLayout linearLayout = new LinearLayout(this);
linearLayout.setLayoutParams(new LinearLayout.LayoutParams(
        ViewGroup.LayoutParams.MATCH_PARENT,
        ViewGroup.LayoutParams.MATCH_PARENT
));
linearLayout.setOrientation(LinearLayout.VERTICAL);
```

6. **Create View Components:**

Create instances of **View** components (e.g., **TextView**, **Button**, etc.) and set their properties.

Java File:

```
TextView textView = new TextView(this);
textView.setText("Hello, World!");
```

```
Button button = new Button(this);
button.setText("Click me");
```

7. **Add Views to the ViewGroup:**

   Add the created **View** components to the **ViewGroup**
   Java File:

```
linearLayout.addView(textView);
linearLayout.addView(button);
```

8. **Set the Content View:**

   Set the **ViewGroup** as the content view of your activity.
   Java File:

```
setContentView(linearLayout);
```

9. **Handle View Events (Optional):**

   If you created interactive components (like buttons), you may want to handle their events programmatically.
   Java File:

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Handle button click
    }
});
```

10. **Run the App:**

    Run your app, and you should see the UI created programmatically.

- **<u>Using Built in Layout Classes: -</u>**

1. Linear Layout: -

In Android, **LinearLayout** is a ViewGroup that arranges its children's elements either horizontally or vertically in a linear fashion. It is one of the basic layout managers provided by the Android SDK and is widely used to create simple and straightforward UIs.

Key attributes of **LinearLayout**: 1. Orientation 2. Weight 3. Gravity

4. Nesting

1. **Orientation:**
   - **android:orientation** attribute determines the arrangement of child views. It can have two values:
     - **"horizontal"**: Arranges child views from left to right.
     - **"vertical"**: Arranges child views from top to bottom (default).

   Example:Xml File –

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <!-- Child views go here -->

</LinearLayout>
```

2. **Weight: android:layout_weight** attribute is used to distribute the available space among child views. It is often used when you want to allocate a specific ratio of space to different views.

```
Example: <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <Button
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 1"
        android:layout_weight="1" />

    <Button
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Button 2"
        android:layout_weight="2" />

</LinearLayout>
```

3. **Gravity:**

- **android:gravity** attribute is used to control the alignment of children within the **LinearLayout**. It affects how the child views are positioned along the axis of the **LinearLayout** (either horizontally or vertically).

```
Example: <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <!-- Child views go here -->

</LinearLayout>
```

4. **Nesting:**

- You can nest **LinearLayouts** within each other or combine them with other layout types (e.g., **RelativeLayout**, **ConstraintLayout**) to create more complex UI structures.

```
Example: <LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

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

        <!-- Horizontal child views go here -->

    </LinearLayout>
    <!-- Additional vertical child views go here -->

</LinearLayout>
```

LinearLayout is easy to use and suitable for simple UIs, but for more complex and flexible layouts, you might explore other layout managers like **RelativeLayout** or **ConstraintLayout** depending on your requirements.

2. Relative Layout: -

In Android development, a Relative Layout is a type of layout manager that allows you to position child views (UI elements) relative to each other or relative to the parent layout. It is part of the Android UI framework and is used in conjunction with XML to define the structure and positioning of user interface elements within an Android app.

Key features and concepts of Relative Layout in Android:

1. **Relative Positioning:** Views within a Relative Layout are positioned relative to each other or to the parent layout. You can specify rules such as above, below, to the left of, to the right of, align top, align bottom, align start, align end, etc.
2. **Attributes:** Relative Layout uses XML attributes to define the relationships between views. Some commonly used attributes include **layout_above**, **layout_below**, **layout_toLeftOf**, **layout_toRightOf**, **layout_alignParentTop**, **layout_alignParentBottom**, **layout_alignParentStart**, and **layout_alignParentEnd**.
3. **Flexible Layouts:** Relative Layout is flexible and allows for dynamic and responsive UI designs. As the size or content of views changes, the positioning rules are applied dynamically, adapting to the changes.

Here's a simple example of a Relative Layout in XML:

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

    android:layout_width="match_parent"

    android:layout_height="match_parent">



    <Button
    android:id="@+id/button1"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="Button 1"

    android:layout_alignParentTop="true"

    android:layout_alignParentStart="true"/>

  <Button

    android:id="@+id/button2"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="Button 2"

    android:layout_below="@id/button1"

    android:layout_alignParentEnd="true"/>

</RelativeLayout>
```

3. Frame Layout: -

A Frame Layout is a type of layout manager in Android that is designed to block out an area on the screen to display a single view. It acts as a simple container for holding a single child element and is often used when you want to show only one item at a time, such as fragments, images, or buttons.

Key features of Frame Layout:

1. **Single Child:** A Frame Layout can have only one direct child. If you add multiple children, they will overlap, and only the topmost child will be visible.
2. **No Distinct Positioning:** Unlike some other layout managers like Relative Layout, Frame Layout does not provide specific attributes for positioning child views relative to each other. Typically, child views are aligned to the top-left corner by default.

3. **Overlap:** When you add multiple child views to a Frame Layout, they will be layered on top of each other. The last child added will be the topmost and visible one.

Here's a simple example of a Frame Layout in XML:

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

    android:layout_width="match_parent"

    android:layout_height="match_parent">

    <!-- Your child views go here -->

    <ImageView

        android:id="@+id/myImageView"

        android:layout_width="match_parent"

        android:layout_height="match_parent"

        android:src="@drawable/my_image" />



</FrameLayout>
```

In this example, the Frame Layout contains an Image View as its child. The Image View takes up the entire space of the Frame Layout. If you add more child views inside the Frame Layout, they will overlap, and only the topmost one will be visible.

Frame Layout is commonly used when you want a simple container to display a single item or to switch between different views dynamically, with only one visible at a time.

4. Table Layout: -

In Android development, a Table Layout is a type of layout manager that organizes child views in rows and columns, similar to an HTML table. It's a way to create a grid-like structure for arranging UI elements in Android applications.

Key features of Table Layout:

1. **Rows and Columns:** Child views are arranged in a grid of rows and columns. Each row can contain multiple views, and each view can span multiple columns or rows.
2. **Table Rows:** Views are typically added to the Table Layout within Table Row elements. Each Table Row represents a row in the table, and you can add child views to it, defining the arrangement of views within that row.
3. **Spanning:** Views can span multiple rows or columns, allowing for flexible and complex layouts.

Here's a simple example of a Table Layout in XML:

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

    android:layout_width="match_parent"

    android:layout_height="match_parent">

    <TableRow>

        <TextView

            android:text="Row 1, Column 1"

            android:padding="10dp"

             android:background="#CCCCCC" />

        <TextView

            android:text="Row 1, Column 2"

            android:padding="10dp"

            android:background="#EEEEEE" />

    </TableRow>

    <TableRow>

        <TextView

            android:text="Row 2, Column 1"

            android:padding="10dp"
            android:background="#CCCCCC" />

        <TextView

            android:text="Row 2, Column 2"

            android:padding="10dp"

            android:background="#EEEEEE" />

    </TableRow>

  </TableLayout>
```

5. Grid Layout: -

A **Grid Layout** in Android is a layout manager that organizes its children in a two-dimensional grid. It is designed to provide a flexible and responsive way to create complex layouts by specifying rows and columns. Each cell in the grid can contain a single view, and views can span multiple rows or columns, enabling the creation of versatile user interfaces.

Here's a brief overview of some key features and attributes of **Grid Layout**:

1. **Rows and Columns:** You can define the number of rows and columns in the grid using the **android:rowCount** and **android:columnCount** attributes.
2. **Cell Spanning:** Child views can span multiple rows or columns using the **android:layout_rowSpan** and **android:layout_columnSpan** attributes.
3. **Grid Spacing:** You can control the spacing between rows and columns using attributes like **android:layout_margin**, **android:layout_rowWeight**, and **android:layout_columnWeight**.
4. **Alignment:** **GridLayout** provides options for aligning its children within cells, such as **android:layout_gravity** and **android:alignmentMode**.
5. **Ordering:** You can control the order in which cells are populated using the **android:columnOrderPreserved** attribute.

Here's a simple example of a **Grid Layout** in XML:

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

   android:layout_width="match_parent"
```

```
android:layout_height="match_parent"

android:columnCount="3"

android:rowCount="3"

android:layout_gravity="center">

<Button

   android:text="Button 1"

   android:layout_columnSpan="1"

   android:layout_rowSpan="1" />

<Button

   android:text="Button 2"

   android:layout_columnSpan="2"

   android:layout_rowSpan="1" />

<Button

   android:text="Button 3"

   android:layout_columnSpan="1"

   android:layout_rowSpan="2" />

<Button

   android:text="Button 4"

   android:layout_columnSpan="1"

   android:layout_rowSpan="1" />

<Button

   android:text="Button 5"

   android:layout_columnSpan="1"
```

```
        android:layout_rowSpan="1" />

    <Button

        android:text="Button 6"

        android:layout_columnSpan="1"

        android:layout_rowSpan="1" />

</GridLayout>
```

In this example:

- The **Grid Layout** is set to have 3 columns and 3 rows.
- Buttons are placed in specific cells, and their spans are defined using **android:layout_columnSpan** and **android:layout_rowSpan**.

Grid Layout is particularly useful for creating dynamic and responsive layouts where the number of rows and columns may vary based on the available space or device characteristics.

It provides a powerful way to create grid-based designs in Android applications.

• Using Container Control Classes: -

In Android, containers are typically represented by layout classes. Layout classes are responsible for organizing and positioning the UI elements (views) within an Android application.

These are classes that are responsible for organizing and managing the layout and positioning of child views within a user interface.

Here are some common container (layout) classes in Android:

1. **Linear Layout:** It arranges child views in a single line either horizontally or vertically.

```
<LinearLayout

    android:layout_width="match_parent"
    android:layout_height="wrap_content"

    android:orientation="vertical">

    <!-- Child views go here -->

</LinearLayout>
```

2. **Relative Layout:** It positions child views relative to each other or relative to the parent layout.

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
 <!-- Child views with relative positioning go here -->
</RelativeLayout>
```

3. **Frame Layout:** It is a simple layout manager that overlays child views on top of each other.

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  <!-- Child views go here -->
</FrameLayout>
```

4. **Grid Layout:** It organizes child views in a two-dimensional grid.

```
<GridLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:rowCount="2"
    android:columnCount="2">
<!-- Child views in a grid go here -->
</GridLayout>
```

5. **Constraint Layout:** It allows you to create complex layouts using constraints to define the position and size of child views relative to each other.

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  <!-- Child views with constraints go here -->
</androidx.constraintlayout.widget.ConstraintLayout>
```

These layout classes act as containers or containers for UI elements, and their use depends on the specific requirements of the user interface you are designing. You often combine these containers and nest them to create more complex and responsive layouts.

-----------------------------XXXXXXXXXXXXXXXXX-----------------------------------