Chapter 2

User Interfaces

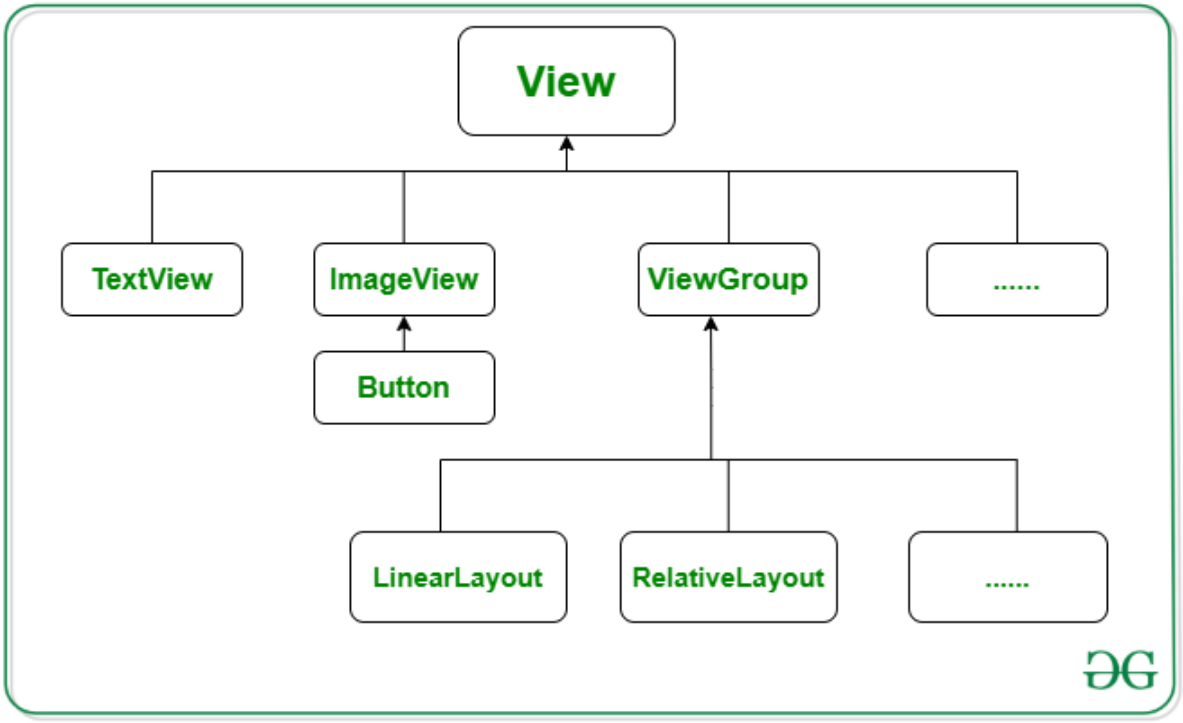
2.1 View

- In Android app development, a **View** is a fundamental building block that represents a UI element or widget that users can interact with or see on the screen.
- Views are used to create the visual interface of an app, allowing users to input data, display information, navigate between screens, and interact with the app's functionality.
- They play a crucial role in presenting the app's user interface and enabling user interactions.
- The View class is the base class or we can say that it is the superclass for all the GUI components in android.
- For example, the *EditText* class is used to accept the input from users in android apps, which is a subclass of View, and another example of the *TextView* class which is used to display text labels in Android apps is also a subclass of View.
- View refer to the **android.view.View** class, which is the base class of all UI classes.
- Following are some of the common View subclasses that will be used in android applications.
  - **TextView**
  - **EditText**
  - **ImageView**
  - **RadioButton**
  - **Button**
  - **ImageButton**
  - **CheckBox**
  - **DatePicker**
  - **Spinner**
  - **ProgressBar**.
- In the given below layout, the **<TextView>** and **<Button>** are two distinct views:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, Android!" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />

</LinearLayout>
```

2.2 View Group

- In Android app development, a **ViewGroup** is a specialized type of View that acts as a container for holding and arranging other Views.
- Unlike regular Views that represent UI elements like buttons or text fields, a ViewGroup doesn't typically display content directly. Instead, it provides a layout structure and arrangement for its child Views.
- The ViewGroup will provide an invisible container to hold other Views or ViewGroups and to define the layout properties.
- For example, Linear Layout is the ViewGroup that contains UI controls like Button, TextView, etc., and other layouts also.
- Following are the commonly used ViewGroup subclasses used in android applications.
  - **FrameLayout**
  - **WebView**
  - **ListView**
  - **GridView**
  - **LinearLayout**
  - **RelativeLayout**
  - **TableLayout** and many more.



View	ViewGroup
View is a simple rectangle box that responds to the user's actions.	ViewGroup is the invisible container. It holds View and ViewGroup
View is the SuperClass of All component like TextView, EditText, ListView, etc	ViewGroup is a collection of Views(TextView, EditText, ListView, etc..), somewhat like a container.
A View object is a component of the user interface (UI) like a button or a text box, and it's also called a widget.	A ViewGroup object is a layout, that is, a container of other ViewGroup objects (layouts) and View objects (widgets)
Examples are EditText, Button, CheckBox, etc.	For example, LinearLayout is the ViewGroup that contains Button(View), and other Layouts also.
View refers to the android.view.View class	ViewGroup refers to the android.view.ViewGroup class
android.view.View which is the base class of all UI classes.	ViewGroup is the base class for Layouts.

2.3 Widget

- In Android app development, a widget refers to a UI element or component that users can interact with on the screen.
- Widgets are used to display information, receive user input, and provide various forms of interaction within an app's user interface.

2.3.1 View vs Widget

- **View**
  - In Android, a View is a fundamental building block of the user interface. It's a base class for all UI components that can be displayed on the screen and interacted with by the user.
  - Views are responsible for drawing themselves on the screen, handling user input, and responding to various events.
  - Examples of Views include buttons, text fields, images, checkboxes, radio buttons, and more.
  - Views can be organized in a hierarchical structure using layout containers like LinearLayout, RelativeLayout, and ConstraintLayout.
  - Views are used to create the visual and interactive elements of an app's user interface.
- **Widget**:
  - A Widget is a specific type of View that represents a UI element designed for a particular purpose, often providing some form of interaction or functionality.
  - Widgets are a subset of Views, and they are typically focused on providing specific actions or displaying specific information.
  - Widgets are often self-contained and designed to be reusable across different parts of an app or even in different apps.
  - Examples of Widgets include buttons, text fields, progress bars, sliders, spinners, switches, and more.
  - The term "widget" is often used to refer to interactive and functional UI elements that are not just passive displays.
- In summary, while both Views and Widgets are fundamental to Android's user interface, Widgets are a subset of Views.
- Views encompass all UI elements, whether interactive or static, while Widgets specifically refer to UI elements that are designed to perform certain actions or provide specific functionality.

2.3.2 Widgets Types

There are given a lot of android widgets with simplified examples such as Button, EditText, CheckBox, ToggleButtonToggleButton, etc.

2.3.2.1 Button

- One common type of widget is the Button, which allows users to trigger actions or navigate to different parts of the app.
- A Button is a UI element that represents a clickable area on the screen. When users tap or click a button, it triggers an action associated with it.
- Buttons are used to perform actions like submitting a form, navigating to another screen, confirming a decision, or initiating a process.
- **Attributes of a Button**: Buttons have various attributes that determine their appearance and behavior
  - **android:text**: Defines the text displayed on the button.
  - **android:id**: Assigns a unique identifier to the button for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the button.
  - **android:onClick**: Specifies the method to be called when the button is clicked.
- **Handling Button Clicks**:
  - To handle button clicks, you can define a method in your activity's Java or Kotlin code. The method should have the same name as the value specified in the **android:onClick** attribute.
  - For example, if you set **android:onClick="onButtonClick"**, you would create a method named **onButtonClick(View view)** in your activity.


```
<Button
    android:id="@+id/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click Me"
    android:onClick="onButtonClick" />
```

- **Button Styles**: Buttons can be customized using styles and themes to match the app's design. You can change their text color, background color, padding, and other visual properties.
- **Button Types**: Android offers various types of buttons, such as standard buttons, image buttons, and floating action buttons (FABs). Each type serves a specific purpose and has a distinct appearance.

  Buttons play a crucial role in guiding users through an app's interface and enabling interactions. By effectively using buttons, you can create intuitive and user-friendly app experiences that allow users to perform actions effortlessly.

2.3.2.2 Edit Text

- In Android app development, an EditText widget is used to provide an area where users can input text.
- It's a versatile widget that allows users to enter single-line or multi-line text, numbers, passwords, and other types of input.
- EditText is commonly used for tasks like user registration, search boxes, comment sections, and more.
- **Attributes of an EditText**:
  - EditText has various attributes that determine its behavior and appearance:
  - **android:id**: Assigns a unique identifier to the EditText for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the EditText.
  - **android:hint**: Sets a hint text that provides a description of the expected input.
  - **android:inputType**: Determines the type of input allowed, such as text, numbers, email, etc.


```
<EditText
    android:id="@+id/inputUsername"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Enter your username"
    android:inputType="text" />
```

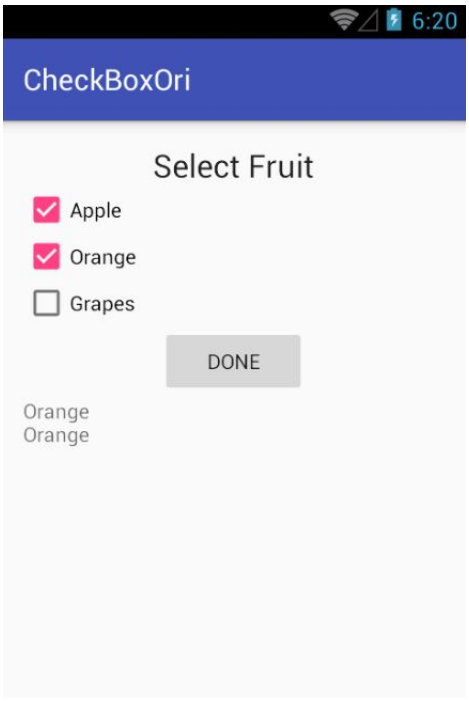
- **Accessing EditText Input**:
  - To access the text entered in an EditText, you can use its ID to reference it in your Java or Kotlin code.
  - Use the **getText()** method to retrieve the text entered by the user.
  - Example Java code:

```
EditText inputUsername = findViewById(R.id.inputUsername);
String username = inputUsername.getText().toString();
```

- **Input Validation**:
  - EditText input can be validated to ensure that users enter the correct format of data (e.g., valid email address, numeric value).
  - You can add listeners to EditText to respond to events like text changes or focus changes.

2.3.2.3 CheckBox

- A CheckBox is a UI element that consists of a square box and a label. Users can tap the box to toggle between checked and unchecked states.
- CheckBoxes are commonly used in forms, settings screens, and other scenarios where users need to make multiple selections.
- CheckBoxes are often used for binary choices (e.g., true/false, yes/no) or multiple selections from a list of options.
- **Attributes of a CheckBox**:
  - **android:id**: Assigns a unique identifier to the CheckBox for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the CheckBox.
  - **android:text**: Sets the label or text displayed next to the CheckBox.
  - **android:checked**: Determines whether the CheckBox is initially checked (true) or unchecked (false).




```
<CheckBox
    android:id="@+id/checkBoxOption"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Option A"
    android:checked="false" />
```

- **Accessing CheckBox State**:
  - To access the state of a CheckBox (whether it's checked or unchecked), you can use its ID to reference it in your Java or Kotlin code.
  - Use the isChecked() method to determine the current state of the CheckBox.
  - Example Java code:

```
CheckBox checkBoxOption = findViewById(R.id.checkBoxOption);
boolean isOptionChecked = checkBoxOption.isChecked();
```

- **Listener for CheckBox Changes**
  - You can add an **OnCheckedChangeListener()** to a CheckBox to respond to changes in its state.

- This listener is invoked when the user checks or unchecks the CheckBox.

```
checkBoxOption.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
            // Handle the CheckBox state change
            }
});
```

2.3.2.4 TextView

- In Android app development, a TextView widget is used to display text content on the screen.
- It can show plain text, formatted text (using HTML-like tags), and even support basic text styling, such as bold, italic, and underline.
- It's a versatile UI element that can be used to show static text, dynamic content, formatted text, and more.
- TextViews are a fundamental component of user interfaces, used for displaying labels, descriptions, instructions, and various forms of textual information.
- **Attributes of a TextView**:
  - **android:id**: Assigns a unique identifier to the TextView for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the TextView.
  - **android:text**: Sets the text content to be displayed in the TextView.
  - **android:textSize**: Specifies the size of the text.
  - **android:textColor**: Sets the color of the text.
  - **android:gravity**: Defines the alignment of the text within the TextView.


```
<TextView
    android:id="@+id/infoTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Welcome to our app!"
    android:textSize="18sp"
    android:textColor="#000000"
    android:gravity="center" />
```

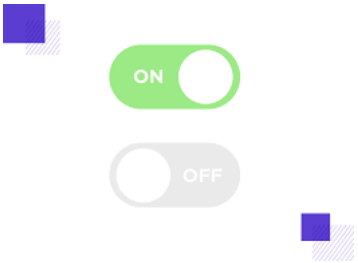
- **Setting Text Programmatically**:
  - You can also set the text of a TextView dynamically in your Java or Kotlin code using its ID.
  - Use the setText() method to set the text content.
  - Example Java code:

```
TextView infoTextView = findViewById(R.id.infoTextView);
infoTextView.setText("Welcome to our app!");
```

- **Text Styling and Formatting**:
  - TextViews can support basic HTML-like tags for text formatting, such as <b> for bold, <i> for italic, and <u> for underline.
  - Use the Html.fromHtml() method to apply formatting to text.

2.3.2.5 Toggle Button

- A ToggleButton is a UI element that looks like a button but functions like a switch.
- It has two states: checked (on) and unchecked (off).
- Users can tap the ToggleButton to switch between the two states, making it suitable for binary choices.
- **Attributes of a ToggleButton**:
  - **android:id**: Assigns a unique identifier to the ToggleButton for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the ToggleButton.
  - **android:textOn** and **android:textOff**: Set the text labels displayed when the ToggleButton is in the on and off states.
  - **android:checked**: Determines whether the ToggleButton is initially in the on (true) or off (false) state.
- Example ToggleButton in XML Layout:




```
<ToggleButton
    android:id="@+id/toggleButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOn="ON"
    android:textOff="OFF"
    android:checked="false" />
```

- **Accessing ToggleButton State**:
  - To access the state of a ToggleButton (whether it's on or off), you can use its ID to reference it in your Java or Kotlin code.
  - Use the **isChecked()** method to determine the current state of the ToggleButton.
  - Example Java code:

```
ToggleButton toggleButton = findViewById(R.id.toggleButton);
boolean isToggleButtonOn = toggleButton.isChecked();
```

- **Listener for ToggleButton Changes**:
  - You can add an OnCheckedChangeListener to a ToggleButton to respond to changes in its state.
  - This listener is invoked when the user toggles the button's state.
  - Example Java code:

```
toggleButton.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
@Override
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
        // Handle the ToggleButton state change
        }
});
```

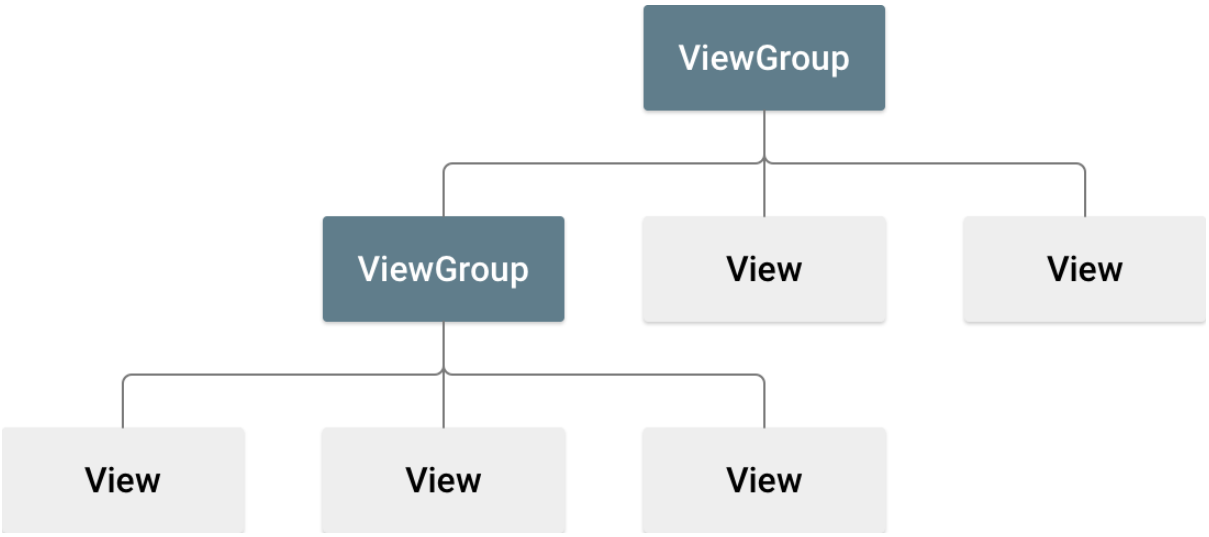
2.4 XML (Extensible Markup Language)

- XML (Extensible Markup Language) is a markup language that is widely used for structuring, storing, and transporting data in a human-readable format.
- In the context of Android app development, XML is used extensively for defining the layout and UI design of app screens.
- It serves as a descriptive way to define the structure and appearance of user interfaces.
- Here's why XML is used in Android app development for layout and UI design:
  - **Declarative Approach**: XML provides a declarative way to define the structure and appearance of UI elements. This means that you describe how you want your UI to look and behave without having to write procedural code to achieve that appearance.
  - **Separation of Concerns**: By using XML for UI design, you separate the presentation layer (layout) from the logic (Java or Kotlin code). This allows developers and designers to work independently on different parts of the app.
  - **Readability**: XML is human-readable, making it easy to understand the structure and hierarchy of UI elements even for those who are not familiar with programming.
  - **Ease of Maintenance**: When you need to modify the layout or UI design, you can simply update the XML layout files without affecting the underlying code logic.
  - **Reusability**: XML layouts are reusable components. You can create a layout once and reuse it across multiple screens or even in different apps.
  - **Consistency**: Using XML layouts helps maintain a consistent look and feel across different parts of your app, contributing to a cohesive user experience.
  - **Tool Integration**: Android development tools like Android Studio provide visual editors that allow you to design UI layouts visually while generating the corresponding XML code in the background.
  - **Responsive Design**: XML layouts support creating responsive designs that adapt well to different screen sizes and orientations.
  - **Internationalization and Localization**: XML layouts can be easily localized to support different languages and cultures.
- Here's an example of an XML layout file defining a simple LinearLayout with a TextView and a Button:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, XML!" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />

</LinearLayout>
```

2.5 Layout

- In Android app development, a layout is a fundamental concept used to define the structure and arrangement of user interface elements (Views) within an activity or fragment.
- Layouts determine how Views are positioned, sized, and aligned on the screen, creating the visual representation of an app's user interface.
- By effectively using layout classes, you can create user interfaces that provide a seamless and user-friendly experience on various devices.
- Layouts are defined in XML layout files located in the res/layout directory of your app.
- A ViewGroup act as a base class for layouts and layouts parameters that hold other Views or ViewGroups and to define the layout properties. They are Generally Called layouts.



2.5.1 Android Layout Types

- Following are the commonly used layouts in android applications to implement required designs.
  - Constraint Layout
  - Linear Layout
  - Relative Layout
  - Frame Layout
  - Absolute Layout
  - Table Layout
  - Scroll View
  - Web View
  - List View
  - Grid View

2.5.2 Linear Layout

- A LinearLayout is one of the basic layout classes in Android app development that arranges child Views in a linear orientation, either horizontally or vertically.
- It's a simple and straightforward layout manager used to create linear arrangements of UI elements.
- LinearLayout is often used when you want to arrange Views in a single row (horizontal) or a single column (vertical).
- **Attributes of LinearLayout**: LinearLayout has several attributes that control its behavior and appearance:
  - **android:id**: Assigns a unique identifier to the LinearLayout for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the LinearLayout. Common values include match_parent and wrap_content.
  - **android:orientation**: Determines the arrangement of child Views. It can be set to "horizontal" for a row arrangement or "vertical" for a column arrangement.
  - **android:gravity**: Specifies the alignment of child Views within the LinearLayout.
  - **android:layout_weight**: Used to distribute available space among child Views based on their weight values.
- **LinearLayout Orientation**: The **android:orientation** attribute specifies how child Views are arranged within the LinearLayout:
  - **"horizontal"**: Child Views are placed side by side in a single row, from left to right.
  - **"vertical"**: Child Views are stacked vertically in a single column, from top to bottom.
- **Gravity and Layout Alignment**: The android:gravity attribute determines how child Views are aligned within the LinearLayout:
  - **"top", "center", "bottom"**: Vertically aligns child Views to the top, center, or bottom, respectively.
  - **"left", "center_horizontal", "right"**: Horizontally aligns child Views to the left, center, or right, respectively.
  - **"center_vertical"**: Vertically centers child Views within the LinearLayout.
  - **"center_horizontal"**: Horizontally centers child Views within the LinearLayout.
- **Layout Weight**:
  - The **android:layout_weight** attribute assigns a weight to child Views. This weight determines how extra space is distributed among child Views. It's especially useful when the LinearLayout's width or height is set to **match_parent**.
  - Child Views with higher weight values will receive more of the available space. For example, if you have three Views with weight values of 1, 2, and 1, the second View will occupy twice as much space as the other two.

- Example of using a vertical LinearLayout to arrange a TextView and a Button

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, LinearLayout!" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me" />

</LinearLayout>
```

2.5.3 Relative Layout

- A RelativeLayout is a layout manager in Android that allows you to create more complex and flexible UI layouts by positioning child Views relative to each other or relative to the parent layout.
- Unlike LinearLayout, which arranges Views linearly in a single direction, RelativeLayout offers more advanced control over the positioning and alignment of UI elements.
- **Attributes of RelativeLayout**: RelativeLayout has attributes that control its behavior and the placement of child Views:
  - **android:id**: Assigns a unique identifier to the RelativeLayout for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the RelativeLayout. Common values include **match_parent** and **wrap_content**.
- **Positioning with Rules**: RelativeLayout allows you to position child Views relative to each other using various layout rules:
  - **android:layout_alignParentTop**, **android:layout_alignParentBottom**, **android:layout_alignParentLeft**, **android:layout_alignParentRight**: Positions a child View at the top, bottom, left, or right edge of the parent RelativeLayout.
  - **android:layout_alignTop**, **android:layout_alignBottom**, **android:layout_alignLeft**, **android:layout_alignRight**: Positions a child View relative to the top, bottom, left, or right edge of another child View.
  - **android:layout_alignStart**, **android:layout_alignEnd**: Positions a child View relative to the start or end edge of another child View (useful for supporting right-to-left languages).
- **Centering**: RelativeLayout also supports centering child Views both horizontally and vertically:
  - **android:layout_centerHorizontal**: Centers a child View horizontally within the parent RelativeLayout.
  - **android:layout_centerVertical**: Centers a child View vertically within the parent RelativeLayout.
  - **android:layout_centerInParent**: Centers a child View both horizontally and vertically within the parent RelativeLayout.

2.5.4 Constraint Layout

- ConstraintLayout is a powerful and flexible layout manager in Android that allows you to create complex UI layouts with a high level of control over the positioning and alignment of UI elements.
- ConstraintLayout uses a constraint-based approach, where UI elements are positioned based on their relationships to other elements and parent layout boundaries.
- Constraint layouts use constraints to define the position and alignment of Views.
- Constraints can be set both horizontally and vertically, offering precise control over the layout.
- **Attributes of ConstraintLayout**:
  - **android:id**: Assigns a unique identifier to the ConstraintLayout for referencing in code.
  - **android:layout_width** and **android:layout_height**: Specify the width and height of the ConstraintLayout. Common values include match_parent and wrap_content.
- **Creating Constraints**: You can define constraints in the XML layout file by setting attributes like **app:layout_constraintTop_toTopOf**, **app:layout_constraintStart_toStartOf**, etc. Some commonly used constraint attributes include:
  - **app:layout_constraintTop_toTopOf**: Aligns the top edge of a View with the top edge of another View or a guideline.
  - **app:layout_constraintEnd_toEndOf**: Aligns the end (right) edge of a View with the end edge of another View or a guideline.
  - **app:layout_constraintBottom_toBottomOf**: Aligns the bottom edge of a View with the bottom edge of another View or a guideline.
  - **app:layout_constraintStart_toStartOf**: Aligns the start (left) edge of a View with the start edge of another View or a guideline.
  - **app:layout_constraintBaseline_toBaselineOf**: Aligns the baseline of a View with the baseline of another View.
- **Chains**: Chains are used to distribute space evenly among a group of Views. You can create horizontal and vertical chains and control the space distribution between elements.
- **Guidelines**: Guidelines help you position Views by providing alignment references. You can create guidelines that are positioned relative to the layout's edges or percentages of the layout's dimensions.
- **Bias**: Bias allows you to control the alignment of a View within its constraints. It defines how much the View is biased toward one end of its constraints.
- **Example**: Here's an example of using a ConstraintLayout to position a TextView above a Button with specific spacing

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Above Button"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintVertical_chainStyle="packed"
        app:layout_constraintBottom_toTopOf="@+id/button"
        android:layout_marginBottom="16dp" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click Me"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginTop="16dp" />

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

2.6 Styles

- In Android app development, styles are a powerful way to define and apply a consistent look and feel to the user interface (UI) elements across your app.
- A style is a collection of attributes that define the appearance of UI components such as TextViews, Buttons, EditTexts, and more.
- Instead of specifying these attributes individually for each UI element, you can group them into a style and apply the style to multiple elements.
- Styles offer several benefits, including:
  - **Consistency**: Styles help ensure a consistent visual identity throughout your app. You can define a single style for a particular type of UI element, and all instances of that element will share the same appearance.
  - **Efficiency**: By using styles, you avoid duplicating attribute values for multiple UI elements. If you need to change the appearance, you can update the style definition, and the changes will be applied to all instances using that style.
  - **Maintainability**: Styles make it easier to manage and maintain your app's visual design. If you decide to make a design change, you can do so in one place (the style) rather than updating individual UI elements.
  - **Ease of Theming**: Android supports theming, which involves creating multiple sets of styles to apply different visual themes to your app. This allows you to give your app a unique look for different purposes or target audiences.
  - **Code Separation**: Styles help separate UI design from code logic. Developers can focus on the functionality while designers can work on creating and modifying styles.
- **Defining Styles**: Styles are typically defined in the **res/values/styles.xml** or **res/values-v21/styles.xml** file (for specific API versions).
- Here's an example of defining a style:

```
<resources>
    <style name="MyButtonStyle" parent="Widget.AppCompat.Button">
        <item name="android:textColor">@color/colorPrimary</item>
        <item name="android:background">@drawable/button_background</item>
        <!-- Other attributes... -->
    </style>
</resources>
```

- In this example, the MyButtonStyle style is defined based on the Widget.AppCompat.Button parent style. It sets the text color and background attributes for the button.
- **Applying Styles**: You can apply a style to a UI element using the style attribute. This can be done directly in XML layout files or programmatically in code.

```
<Button
    android:id="@+id/myButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/MyButtonStyle"
    android:text="Click Me" />
```

2.7 Themes

- In Android app development, themes are a collection of styles that define the overall look and feel of an app's user interface (UI).
- A theme encapsulates the visual design aspects of your app, including colors, fonts, styles, and more.
- By applying a theme to your app, you can easily maintain a consistent and cohesive appearance across all UI elements.
- Themes are an essential part of Android app design as they allow you to create a unified and visually appealing user experience.
- Here's a detailed explanation of themes in Android:
  - **Styles and Attributes**: Themes are made up of styles, which are collections of attribute-value pairs that define the appearance of UI elements. Attributes can include text color, background color, font, and other visual properties.
  - **Inheritance**: Themes support inheritance, which means you can define a parent theme and then create child themes that inherit attributes from the parent. This makes it easy to create variations of a theme while maintaining a consistent base style.
  - **Component-Level Theming**: Themes allow you to define styles for specific components (such as buttons, text views, and more) as well as global styles that apply to the entire app.
  - **Resource Files**: Theme definitions are usually stored in XML files, such as res/values/themes.xml or res/values/styles.xml. Android provides a set of predefined attributes that you can use to customize your theme.
- Creating Theme:
  - **Defining Themes**: Themes are defined in XML files with the .xml extension. You typically create a styles.xml file in the res/values directory to define your app's themes.
  - Here's an example of defining a basic theme:

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
        <!-- Other attributes... -->
    </style>
</resources>
```

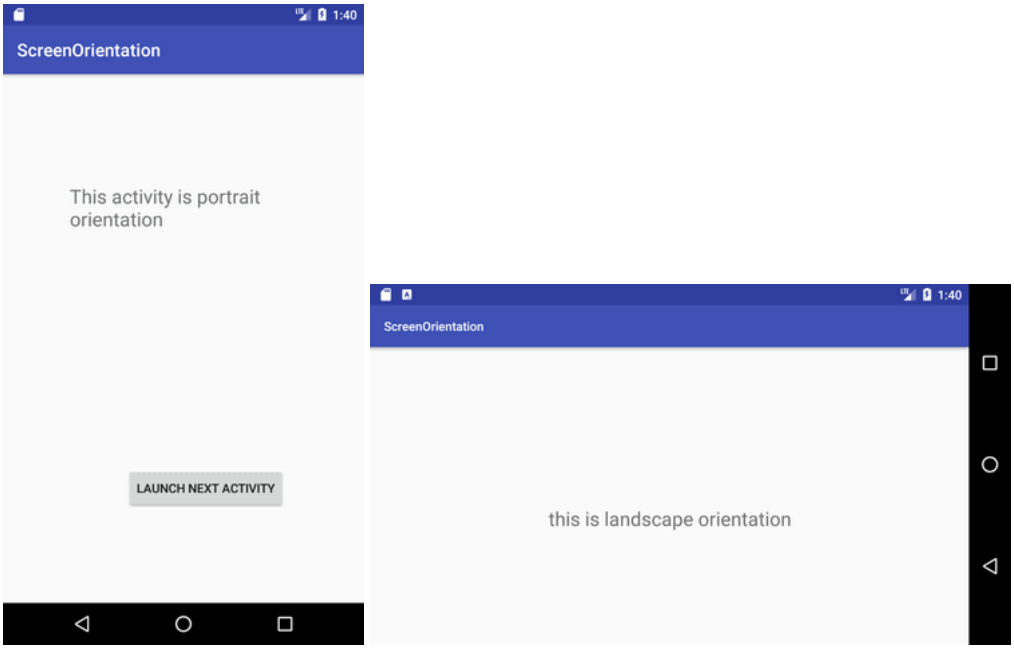
  - In this example, the AppTheme theme is based on the **Theme.AppCompat.Light** parent theme. It sets the primary, primary dark, and accent colors.
- Applying Themes:
  - You can apply a theme to your app in the AndroidManifest.xml file using the **android:theme attribute**.

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
        <!-- Other attributes... -->
    </style>
</resources>
```

- **Advantages of Themes**:
  - **Consistency**: Themes ensure a consistent design across the entire app, making it visually appealing and user-friendly.
  - **Customization**: You can easily customize the look and feel of your app by modifying theme attributes.
  - **Branding**: Themes allow you to create branded apps with unique color schemes, fonts, and visual elements.
  - **Efficiency**: By applying a theme, you can avoid repetitive styling code and streamline the design process.
  - **Easy Updates**: If you need to change the app's design, you can do so by updating the theme definition, affecting the entire app.
- Themes are a crucial aspect of Android app development that enable you to create a consistent and visually appealing design throughout your app.
- Themes provide a centralized way to manage the appearance of UI elements and enhance the user experience.

2.8 Orientation

- In Android Studio development, orientation refers to the physical orientation of a device's screen.
- Android devices have the ability to detect their orientation, which can be either portrait or landscape.
- The orientation of the device impacts how the user interface (UI) is displayed and how an app responds to changes in orientation.



- Here's a detailed explanation of orientation in Android Studio development:
- **Portrait Orientation**:
  - Portrait orientation is the default and most common orientation for devices. In this mode, the device's screen is taller than it is wide.
  - Apps designed for portrait orientation are optimized to be used vertically, with UI elements stacked one on top of another.
  - Portrait orientation is suitable for reading, scrolling through content, and apps that are primarily used in a portrait layout.
- **Landscape Orientation**:
  - Landscape orientation occurs when the device is rotated horizontally, making the screen wider than it is tall.
  - Apps designed for landscape orientation take advantage of the wider screen space and can display more content side by side.
  - Landscape orientation is often used for activities like watching videos, playing games, and using apps that benefit from a wider display.
- **Handling Orientation Changes**: When the user rotates the device, the orientation changes, and Android apps need to respond accordingly to provide a smooth user experience. Here's how orientation changes are typically handled:
  - **Configuration Changes**: When the orientation changes, Android may trigger a configuration change event. This can cause the activity to be recreated, which can lead to the loss of data or UI state if not properly managed.
  - **Handling Configuration Changes**: To handle configuration changes like orientation, you can override the **onSaveInstanceState()** method to save important data and restore it in the **onCreate()** method.

- **Layout Variants**: You can create separate layouts for portrait and landscape orientations to ensure that your app's UI looks and functions well in both modes. These layouts are stored in the res/layout and res/layout-land directories, respectively.
- **Locking Orientation**: You can choose to lock the orientation of an activity to either portrait or landscape mode programmatically or through XML attributes to prevent orientation changes.

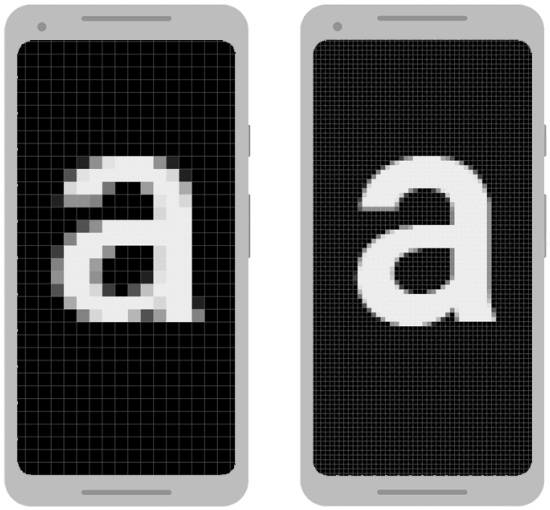
  Handling orientation changes properly is crucial for providing a seamless user experience and ensuring that your app's UI looks and functions well in different orientations.

2.9 Screen Size and Screen Density

- In Android application development, screen size and screen density are important concepts that help developers design user interfaces (UIs) that adapt well to different devices.
- These concepts take into account the physical size and resolution of the device's screen, ensuring that the app's layout, fonts, and images are displayed correctly and consistently across a variety of devices.
- **Screen Size**: Screen size refers to the physical dimensions of a device's screen. It's typically measured diagonally in inches. Android categorizes devices into four general screen size groups:
  - **Small Screens**: Devices with small screens, such as older Android phones with smaller displays.
  - **Normal Screens**: Most Android devices fall into this category. It includes a wide range of phone sizes.
  - **Large Screens**: Tablets and devices with larger displays are considered to have large screens.
  - **Extra-Large Screens**: This category includes larger tablets and devices with very large screens.
- **Screen Density**: Screen density, also known as pixel density, refers to the number of pixels that fit within a certain area of the screen. It's usually measured in pixels per inch (PPI) or dots per inch (DPI). Android devices are categorized into several density buckets:
  - **Low Density (LDPI)**: Devices with lower pixel densities. This category is rarely used in modern devices.
  - **Medium Density (MDPI)**: Devices with a baseline density of 160 DPI.
  - **High Density (HDPI)**: Devices with higher pixel densities, typically around 240 DPI.
  - **X-High Density (XHDPI)**: Devices with even higher pixel densities, around 320 DPI.
  - **XX-High Density (XXHDPI)**: Devices with very high pixel densities, around 480 DPI.
  - **XXX-High Density (XXXHDPI)**: Devices with extremely high pixel densities, around 640 DPI.

2.10 Unit of Measurement

- In Android development, there are several units of measurement that are used to define the size and positioning of UI elements.
- These units help ensure that your app's layout remains consistent across different screen sizes and densities.
- Here's an explanation of the common units of measurement:
- **px (Pixels)**:
  - A pixel is the smallest unit of display on a screen.
  - In Android, dimensions specified in pixels are considered absolute and are not affected by the device's screen density.
  - Using pixels directly can lead to inconsistent layouts on different devices with varying screen densities.
- **dp (Density-independent Pixels)**:
  - dp is a unit of measurement that takes into account the screen density of the device.
  - 1 dp is equal to 1 pixel on a 160 dpi (MDPI) screen. The system scales dp values based on the screen's density to provide a consistent size.
  - Use dp for specifying dimensions in your layout to ensure that UI elements have a consistent physical size across different devices.
  - Avoid using pixels to define distances or sizes. Defining dimensions with pixels is a problem because different screens have different pixel densities, so the same number of pixels corresponds to different physical sizes on different devices.



  - Consider the two devices above figure. A view that is 100 pixels wide appears much larger on the device on the left. A view defined to be 100 dp wide appears the same size on both screens.
- **sp (Scaled Pixels)**:
  - sp is similar to dp, but it also takes into account the user's preferred font size settings.
  - It's used for specifying text sizes to ensure that text remains readable and accessible, even if the user has set a larger font size.
  - When defining text sizes, you can instead use scalable pixels (sp) as your units. The sp unit is the same size as a dp, by default, but it resizes based on the user's preferred text size. Never use sp for layout sizes.

For example, to specify spacing between two views, use dp:

```
<Button android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/clickme"
    android:layout_marginTop="20dp" />
```

When specifying text size, use sp:

```
<TextView android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="20sp" />
```

- **dpi (Dots Per Inch)**:
  - DPI refers to the density of pixels on a screen and is usually expressed as "dots per inch."
  - Different devices have different screen densities, such as LDPI, MDPI, HDPI, XHDPI, XXHDPI, and XXXHDPI. Each density corresponds to a specific number of pixels per inch.
  - Images and graphics should be provided in different resolutions to ensure they appear crisp on various devices.
- **pt (Points)**:
  - Points are a unit of measurement often used in typography and print design.
  - In Android, 1 pt is roughly equivalent to 0.75 dp. Points are primarily used for specifying text sizes in XML layouts.

2.10.1 Convert dp units to pixel units

In some cases, you need to express dimensions in dp and then convert them to pixels. The conversion of dp units to screen pixels is as follows:

```
px = dp * (dpi / 160)
```

> ★ **Note:** Never hardcode this equation to calculate pixels. Instead, use `TypedValue.applyDimension()`, which converts many types of dimensions (dp, sp, etc.) to pixels for you.

Imagine an app in which a scroll or fling gesture is recognized after the user's finger has moved at least 16 pixels. On a baseline screen, a user's finger must move 16 pixels / 160 dpi, which equals 1/10 of an inch (or 2.5 mm), before the gesture is recognized.

On a device with a high-density display (240 dpi), the user's finger must move 16 pixels / 240 dpi, which equals 1/15 of an inch (or 1.7 mm). The distance is much shorter, and the app therefore appears more sensitive to the user.

To fix this issue, express the gesture threshold in code in dp and then convert it to actual pixels. For example:

Kotlin Java

```
// The gesture threshold expressed in dp
private final float GESTURE_THRESHOLD_DP = 16.0f;

// Convert the dps to pixels, based on density scale
int gestureThreshold = (int) TypedValue.applyDimension(
  COMPLEX_UNIT_DIP,
  GESTURE_THRESHOLD_DP + 0.5f,
  getResources().getDisplayMetrics());

// Use gestureThreshold as a distance in pixels...
```

The `DisplayMetrics.density` field specifies the scale factor used to convert dp units to pixels according to the current pixel density. On a medium-density screen, `DisplayMetrics.density` equals 1.0, and on a high-density screen it equals 1.5. On an extra-high-density screen, it equals 2.0, and on a low-density screen, it equals 0.75. This figure is used by `TypedValue.applyDimension()` to get the actual pixel count for the current screen.

2.11 Sample Questions

Marks	Question
2	How well do Android apps work on devices with different screen sizes? Give an explanation for your answer.
4	In Android UI design, how important are views, view groups, and widgets?
6	Analyse the pros and cons of using various screen layouts when creating Android apps.
6	Analyse how styles and themes help maintain a unified aesthetic across different screens in an Android app.
6	Methodologies for handling device orientation changes in Android apps are solicited.
3	The terms "px," "dp," "sp," "dpi," and "pt" are all used in Android programming; please define them.
3	Explain how Android does the conversion from density-independent pixels (dp) to pixels (px) and give an example.
3	Determine what aspects of an Android app can affect its responsiveness across a range of screen sizes.
9	Examine how the use of various units of measurement (px, dp, sp, dpi, pt) affects the layout of Android app user interfaces.
4	The "Activity" class in Android serves a number of purposes, but its principal one is to interact with the user interface.
8	Analyse the benefits of utilising various layouts (such as LinearLayout and RelativeLayout) while creating user interfaces for Android apps.
8	In the context of Android UI design, discuss the similarities and differences between the "Button" and "ToggleButton" widgets.
8	Analyse how styles and themes help keep an Android app looking consistent.
12	Analyse the difficulties programmer's have while making Android apps responsive to different screen sizes and resolutions.
12	Come up with a strategy for developing a landscape- and portrait-compatible Android app.
5	Explain in detail how to process data from Android's "EditText" and "CheckBox" widgets.
10	Examine the factors that should be thought about when choosing a colour scheme for an Android app.
15	Analyse how including accessibility features into an Android app's design changes the user experience.
6	Detailed instructions for building an Android app's custom View are needed.
12	Explain the benefits and drawbacks of using "ViewModel" for managing UI-related data in Android programming.

2.12 Related Videos

- **XML**
  - https://www.youtube.com/watch?v=UtaY0Jg01pE
- **View and ViewGroup**
  - https://www.youtube.com/watch?v=0TtJwa5vL6Y
- **Linear Layout and Relative Layout**
  - https://www.youtube.com/watch?v=PeC0KgAua7A
  - **Detailed Video**: https://www.youtube.com/watch?v=a3Y0xAdBmLw
  - **Detailed Video**: https://www.youtube.com/watch?v=QdGzmVIYsbA
- **Constraint Layout**
  - **Constraint Layout (Basics)**: https://www.youtube.com/watch?v=VsgXFdynDuQ
  - **Constraint Layout (Chains and Guidelines)**: https://www.youtube.com/watch?v=PqkWT92BT3U
  - **Detailed Video**: https://www.youtube.com/watch?v=q6MxqtcCdl0
- **Style and Themes**
  - https://www.youtube.com/watch?v=DeyFygSs9Qc
- **Orientation**
  - https://www.youtube.com/watch?v=KJGKj078Qag
- **Screen Resolution or Unit of measurement**
  - https://www.youtube.com/watch?v=3tzEcB-G0KI
  - https://www.youtube.com/watch?v=o0995NBtCm0