

Android User Interface Elements

Study Guide

Mr. Akash Suresh Patil
Assistant Professor
CSE, Parul Institute of Technology
Parul University

Contents

1. Android User Interface Elements	Error! Bookmark not defined.1
1.1 Introduction of Material Design.....	3
1.2 UI and UX Layouts: Linear Layout	4
1.3 Absolute Layout, Frame Layout, Relative Layout, Constraint Layout.....	6.
1.4 Dynamic Implementation of Layout	8
1.5 UI widgets with properties.....	10
1.6 events and methods.....	12
1.7 Dialog boxes, Menus: Option and Context.....	14

1. Android User Interface Elements

1.1 Introduction of Material Design

Material Design, a comprehensive design language developed by Google, was **introduced on June 25, 2014**, at the Google I/O conference. It aimed to create a unified and consistent user experience across Google's array of web and mobile products, including Android, by synthesizing classic design principles with technological innovation.

Key Principles

Material Design is built on three core principles:

- **Material is the Metaphor:** The design is inspired by the physical world of paper and ink, with digital surfaces behaving like sheets of material that can expand and reform intelligently. This concept is supported by the use of elevation (z-axis), subtle shadows, and defined edges, which give elements a sense of depth and hierarchy.
- **Bold, Graphic, Intentional:** It leverages foundational elements of print-based design such as robust typography (initially the Roboto typeface), grid-based layouts, intentional use of white space, scale, color, and imagery to create focus and meaning.
- **Motion Provides Meaning:** Motion and animation are not merely decorative; they guide the user's focus, provide feedback for interactions (e.g., ripple effects when a button is tapped), and maintain a sense of continuity as the UI transforms.

Impact on Android User Interface Elements

Material Design brought a significant shift from the previous dark "Holo" theme to a lighter, more vibrant, and structured interface. Its impact on Android UI elements includes:

- **New and Enhanced Components:** Google introduced specific components and widgets, available through the [Material Components library](#), to help developers implement the new design language consistently. Key elements include:
 - **Floating Action Button (FAB):** A circular button that floats above the UI, representing the primary action on a screen.
 - **Cards:** Versatile containers (using the CardView widget) that display content and actions on a single subject, leveraging elevation and shadows for a "paper" look.
 - **App Bars/Toolbars:** Replaced the older Action Bars, offering more flexibility for branding, navigation, and actions at the top of the screen.
 - **Snackbars and Dialogs:** Standardized components for communication, alerting users to information or messages.
 - **Navigation Drawers and Tabs:** Provided consistent patterns for app navigation.
- **Consistent Visual Language:** Material Design promoted a unified and predictable look across apps, making the Android ecosystem feel more cohesive and professional compared to the fragmentation of earlier versions.
- **Emphasis on Accessibility and Customization:** Later iterations, like Material Design 2 (Material Theming, 2018) and Material Design 3 (Material You, 2021), have placed a greater focus on brand customization (color, typography, shape) and dynamic color, which can adapt to a user's wallpaper, while ensuring accessibility guidelines (e.g., sufficient color contrast) are met.

Overall, Material Design provided a robust, rule-based framework that transformed the Android UI into an expressive, intuitive, and visually appealing experience.

1.2 UI and UX Layouts: Linear Layout

The **LinearLayout** is a fundamental ViewGroup in Android UI development that organizes all its child elements in a **single direction**, either **vertically** or **horizontally**. It is a simple and efficient layout for basic, sequential arrangements of UI components.

Key Concepts and Attributes

The behavior of a LinearLayout is primarily defined by its attributes in the XML layout file or by methods in Kotlin/Java code:

Attribute	Description
android:orientation	(Required) Defines the direction in which child views are arranged. It accepts two values: "vertical" (children are stacked in one column) or "horizontal" (children are stacked in one row). The default is horizontal if not specified.
android:layout_weight	(Assigned to children) An "importance" value that specifies how much of the <i>remaining</i> space a child view should occupy within the parent LinearLayout. This is crucial for creating responsive layouts that adapt to different screen sizes.
android:weightSum	(Assigned to the LinearLayout) Defines the maximum total weight sum of all children within the layout. If not specified, the sum is calculated automatically based on the children's declared weights.
android:gravity	(Assigned to the LinearLayout) Controls the alignment of <i>all</i> child views within the parent layout's own boundaries (e.g., center, bottom, right).
android:layout_gravity	(Assigned to a child view) Specifies how a <i>single</i> child view should be aligned within its parent LinearLayout, but only in the direction <i>opposite</i> to the layout's orientation. For a vertical LinearLayout, layout_gravity affects horizontal position (left, center, right); for a horizontal LinearLayout, it affects vertical position (top, center, bottom).

How to Use layout_weight Effectively

The layout_weight attribute is a powerful tool for proportional space distribution:

- **Equal Distribution:** To make multiple views share the space equally, set the android:layout_width of each view to "0dp" (for horizontal orientation) or android:layout_height to "0dp" (for vertical orientation), and set the android:layout_weight of each to "1".
- **Unequal Distribution:** You can assign different weight values to create proportional divisions. The space remaining after all views with a default weight of 0 (wrap_content) are measured is distributed among the views with assigned weights.

Use Cases and Considerations

- **Ideal for Simple Structures:** LinearLayout is best for straightforward UIs like forms, lists, or

toolbars where a sequential arrangement is needed.

- **Performance:** LinearLayouts are generally efficient in terms of performance because they involve less complex calculations during the layout and measure stages compared to more dynamic layouts.
- **Nesting:** While you can nest LinearLayouts to achieve more complex designs, excessive nesting can lead to performance issues. For complex, non-linear layouts, **ConstraintLayout** is the recommended alternative for better performance and tooling support.

1.3 Absolute Layout, Frame Layout, Relative Layout, Constraint Layout

Android provides several ViewGroup subclasses (layouts) to organize user interface elements. The primary layouts, each serving a different purpose, are the **FrameLayout**, **RelativeLayout**, and the modern **ConstraintLayout**. The **AbsoluteLayout** is a deprecated legacy component.

AbsoluteLayout

The **AbsoluteLayout** allows you to specify the exact location of child views within the parent view using **X and Y coordinates**.

- **Characteristics:** It provides precise, pixel-based positioning.
- **Status:** It is **deprecated** and highly discouraged for use in modern Android development.
- **Reason for Deprecation:** It does not adapt well to different screen sizes, resolutions, and aspect ratios. UIs built with **AbsoluteLayout** often appear misaligned or broken on devices with different dimensions than the one they were designed for.
- **Alternatives:** **RelativeLayout** and **ConstraintLayout** provide flexible, screen-independent positioning methods.

FrameLayout

The **FrameLayout** is a simple layout that is designed to block out an area on the screen to display a **single child view**.

- **Characteristics:**
 - Children are drawn in a stack, with the most recently added view appearing on top (LIFO - Last In First Out).
 - The size of the **FrameLayout** is determined by its largest child.
 - Child positioning can be controlled using the `android:layout_gravity` attribute.
- **Use Cases:**
 - Displaying a single dynamic view (e.g., a **ProgressBar** or a single image).
 - Acting as a placeholder for fragments that are swapped in and out during navigation.
 - Overlaying views (e.g., placing text over an image) using gravity and margins.

RelativeLayout

The **RelativeLayout** allows you to position UI elements based on **relationships between sibling views** or the **parent container**.

- **Characteristics:**
 - Positions are specified using rules like `android:layout_below="@+id/other_view"`, `android:layout_alignParentRight="true"`, or `android:layout_centerInParent="true"`.
 - It can help create a flat view hierarchy, which improves performance by reducing nested layouts.
- **Use Cases:**
 - Complex UIs where elements need to be aligned relative to one another or the screen edges (e.g., a username and password field, a "Sign In" button below them, centered horizontally).
- **Note:** The official Android documentation recommends using **ConstraintLayout** instead for better performance and tooling support.

Constraint Layout

The **Constraint Layout** is the **recommended and most flexible** layout for building responsive UIs in Android.

- **Characteristics:**
 - It operates similarly to **RelativeLayout** but offers more advanced alignment and

positioning tools.

- It allows you to define constraints for both vertical and horizontal axes, linking a view's edges to other elements or guidelines.
- It helps in creating large and complex layouts with a flat view hierarchy, which is crucial for good performance across various devices.
- It is deeply integrated with the Android Studio Layout Editor, making UI design intuitive with drag-and-drop functionality and snap-to-constraint features.

- **Use Cases:**

- Virtually all modern Android UI designs, from simple screens to complex, dynamic interfaces that must adapt to different screen sizes and orientations.

Layout Type	Description	Primary Use Case	Status/Recommendation
AbsoluteLayout	Positions elements using fixed X and Y coordinates.	Should not be used.	Deprecated
FrameLayout	Displays a single child view or stacks multiple children for overlay effects.	Fragments, single dynamic elements, overlays.	Active, for specific use cases
RelativeLayout	Positions elements relative to siblings or the parent container.	Complex, flexible layouts (legacy).	Active, but ConstraintLayout is recommended
ConstraintLayout	The most flexible layout, using constraints to define element positions.	Modern, responsive, and complex UIs.	Recommended for all new development

1.4 Dynamic Implementation of Layout

Dynamic implementation of layouts in Android involves creating or modifying UI elements **programmatically during runtime** using Kotlin or Java code, rather than defining everything statically in XML layout files. This approach is essential for UIs that need to adapt based on user input, data fetched from a database or network, or varying screen configurations.

Core Concepts

- **View and ViewGroup Hierarchy:** Android UIs are built as a hierarchy of View objects (widgets like Button, TextView) and ViewGroup objects (layouts like LinearLayout, ConstraintLayout, which contain other views). Dynamic implementation involves manipulating this hierarchy.
- **Context:** When creating views programmatically, you need a Context, usually the current Activity instance (this within an Activity class), to access system resources and themes.
- **LayoutParams:** Every view added programmatically **must** have LayoutParams set. These parameters instruct the parent ViewGroup on how to size and position the child view (e.g., MATCH_PARENT, WRAP_CONTENT, margins, weights). The specific LayoutParams subclass must match the parent layout (e.g., LinearLayout.LayoutParams for a LinearLayout).
- **addView() Method:** Views are added to a ViewGroup using the addView() method, often within an Activity's onCreate() method or in response to a user action.

Dynamic Implementation with LinearLayout (Example in Kotlin)

Here is how to dynamically create TextView elements and add them to a LinearLayout already defined in XML.

1. XML Layout (e.g., activity_main.xml)

xml

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

2. Kotlin Code (e.g., MainActivity.kt)

kotlin

```
import android.os.Bundle
```

```
import android.view.ViewGroup
import android.widget.LinearLayout
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // 1. Get a reference to the parent layout container from the XML
        val linearLayout: LinearLayout = findViewById(R.id.dynamic_layout_container)

        // 2. Loop to create multiple views dynamically
        for (i in 0 until 5) {
            // 3. Instantiate a new view
            val textView = TextView(this)

            // 4. Set layout parameters (crucial step)
            val layoutParams = LinearLayout.LayoutParams(
                ViewGroup.LayoutParams.MATCH_PARENT, // Width
                ViewGroup.LayoutParams.WRAP_CONTENT // Height
            )
            textView.layoutParams = layoutParams

            // 5. Set other properties (text, ID, etc.)
            textView.text = "Dynamic Text View #$i"
            textView.id = View.generateViewId() // Generate a unique ID if needed
            // for later reference or constraints

            // 6. Add the new view to the parent layout
            linearLayout.addView(textView)
        }
    }
}
```

Dynamic Implementation with ConstraintLayout

Adding views dynamically to a ConstraintLayout is more complex because you must also programmatically define all necessary constraints to position the view correctly. This is often done using a ConstraintSet.

```
// ... inside your Activity ...

val constraintLayout: ConstraintLayout = findViewById(R.id.constraint_container)
val newButton = Button(this)
newButton.text = "Dynamic Button"
newButton.id = View.generateViewId() // Must have a unique ID

// 1. Add the view to the layout *before* cloning the ConstraintSet
constraintLayout.addView(newButton)

// 2. Define constraints using a ConstraintSet
val constraintSet = ConstraintSet()
constraintSet.clone(constraintLayout) // Clone existing constraints first

// Connect constraints (e.g., center horizontally and align top to parent's top with a margin)
constraintSet.connect(newButton.id, ConstraintSet.TOP, ConstraintSet.PARENT_ID,
    ConstraintSet.TOP, 100)
constraintSet.connect(newButton.id, ConstraintSet.START, ConstraintSet.PARENT_ID,
    ConstraintSet.START, 0)
constraintSet.connect(newButton.id, ConstraintSet.END, ConstraintSet.PARENT_ID,
    ConstraintSet.END, 0)

// 3. Apply the constraints to the layout
constraintSet.applyTo(constraintLayout)
```

Best Practices

- **Use RecyclerView for Lists:** For dynamic lists of data, always prefer RecyclerView over adding many views to a LinearLayout manually. RecyclerView efficiently recycles views, improving performance and memory usage.
- **Use LayoutInflater for Complex Views:** If your dynamic element is a complex view hierarchy (e.g., a custom card designed in its own XML file), use LayoutInflater to inflate that XML into a View object first, then add the inflated view to your parent layout.
- **Assign Unique IDs:** Views created programmatically do not automatically get an ID. Use View.generateViewId() to assign a unique ID if you need to reference the view later or use it in a ConstraintLayout

1.5 UI widgets with properties

Android UI widgets, also known as views, are the fundamental, interactive building blocks of an application's user interface. Each widget has a unique set of properties (XML attributes) that control its appearance and behavior.

Here are common Android UI widgets and their key properties:

Common View Properties

Most UI widgets share a common set of attributes inherited from the base View class:

Attribute	Description	Common Values
android:id	A unique identifier for the view, necessary for referencing it in code.	@+id/my_element_id
android:layout_width	Specifies the width of the view.	"match_parent", "wrap_content", "100dp"
android:layout_height	Specifies the height of the view.	"match_parent", "wrap_content", "50dp"
android:visibility	Controls whether the view is displayed on the screen.	"visible", "invisible", "gone"
android:background	Defines the background color or drawable for the view.	"#RRGGBB", "@drawable/my_background"
android:padding	Space between the view's content and its boundaries.	"10dp"
android:layout_margin	Space between the view and surrounding elements/parent.	"10dp"

Specific UI Widget Properties

The table below lists the primary widgets and their specific, essential properties:

Widget	Description	Key Properties (XML Attributes)
TextView	Displays non-editable text to the user.	android:text (the actual text), android:textSize (size in sp), android:textColor (color), android:textStyle (bold/italic)
Button	A clickable element that triggers an action when pressed.	android:text (button label), android:onClick (method to call when clicked), android:enabled (true/false)
EditText	A flexible text field for user input.	android:hint (placeholder text when empty), android:inputType (text, number, password, etc.), android:maxLines (max number of lines)

ImageView	Displays images or icons.	android:src or app:srcCompat (source drawable), android:scaleType (how the image is sized/positioned within the view, e.g., centerCrop, fitCenter)
CheckBox	Allows a user to select zero or multiple options from a list.	android:text (label), android:checked (initial state: true/false), android:onCheckedChangeListener (listener method)
RadioButton	Allows a user to select exactly one option within a RadioGroup.	android:text (label), android:checked (initial state: true/false). Must be inside a RadioGroup
ProgressBar	Indicates progress of an operation (spinning wheel or horizontal bar).	android:indeterminate (true for endless progress), android:max (for horizontal bar), style (e.g., ?android:attr/progressBarStyleLarge)
SeekBar	A slider allowing the user to select a value within a range.	android:max, android:progress (current value), android:thumb (custom drawable for the drag handle)

1.6 Events and methods

- Android UI elements are interactive components whose behavior is controlled through a combination of **events** (user interactions like taps, swipes, or text entry) and **methods** (functions you call on UI objects to programmatically change their state, appearance, or behavior).
- **Handling User Events**
- Events are typically captured and handled using *listeners* or *callback interfaces* that you attach to UI components.
- **1. Click Events (Button, ImageButton, TextView, etc.)**
- This is the most frequent event, triggered when a user taps a view.
- **Methods and Listeners:**
- **setOnClickListener(listener):** The primary method to attach a listener in code.
- **onClick(View v):** The callback function within the listener interface that executes when the view is clicked.
- **Programmatic Implementation (Standard Kotlin):**

```
• kotlin

• val myButton: Button = findViewById(R.id.my_button)

• // Using a lambda function as the listener

• myButton.setOnClickListener { view ->

•     // Code to execute when the button is clicked

•     view.setBackgroundColor(Color.GRAY)
```

- }

- **XML Implementation (Simplified):**

- You can define a public method in your Activity and link it in the XML using the android:onClick attribute:

- XML: <Button android:onClick="onSaveButtonClick" ... />

- Kotlin:

- kotlin

- *// This method must be public and accept a 'View' parameter*

- fun onSaveButtonClick(view: View) {

- *// Code to execute when the button is clicked*

- }

-

- **2. Text/Input Events (EditText)**

- These events track changes to text entered by the user in real-time.

- **Method and Listener:**

- **addTextChangedListener(listener):** Registers a listener to monitor text changes.

- **onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int):** Called *every time* the text changes.

kotlin

```
val editText: EditText = findViewById(R.id.my_edit_text)
```

```
editText.addTextChangedListener(object : TextWatcher {
```

```
    override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after: Int) {
```

```
        // Called before the text changes
```

```
    }
```

```
    override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count: Int) {
```

```
        // Update UI as the user types
```

```
    }
```

```
    override fun afterTextChanged(s: Editable?) {
```

```
        // Called after the change is applied (useful for final validation)
```

```
    }
```

})

- **3. Check/State Change Events (CheckBox, RadioButton, Switch)**
- These events fire when the checked state of a component changes.
- **Method and Listener:**
- **setOnCheckedChangeListener(listener):** Registers the listener.
- **onCheckedChanged(buttonView: CompoundButton, isChecked: Boolean):** The callback that provides the current state (isChecked is true or false).

kotlin

```
val checkBox: CheckBox = findViewById(R.id.my_checkbox)

checkBox.setOnCheckedChangeListener { buttonView, isChecked ->
    if (isChecked) {
        // Checkbox is now checked
    } else {
        // Checkbox is now unchecked
    }
}
```

- **Key Methods for Modifying UI Elements Programmatically**
- You use methods to dynamically change the state, appearance, or visibility of views in your Kotlin/Java code during runtime.

• Method	• Description	• Example (Kotlin)
• setText()	• Changes the text displayed in a TextView or Button.	• myTextView.text = "New Message"
• setVisibility()	• Hides or shows a view.	• myView.visibility = View.GONE (Hides and removes space)
• setEnabled()	• Enables or disables user interaction with the view.	• myButton.isEnabled = false
• setBackgroundColor()	• Changes the background	• myView.setBackgroundColor(Color.RED)

	color of the view.	
<ul style="list-style-type: none"> setImageResource() 	<ul style="list-style-type: none"> Changes the image source of an ImageView using a resource ID. 	<ul style="list-style-type: none"> <code>myImageView.setImageResource(R.drawable.new_icon)</code>
<ul style="list-style-type: none"> addView() / removeView() 	<ul style="list-style-type: none"> Dynamically adds or removes child views from a ViewGroup (Layout). 	<ul style="list-style-type: none"> <code>myLinearLayout.addView(newButton)</code>
<ul style="list-style-type: none"> requestFocus() 	<ul style="list-style-type: none"> Forces a view to become the active focus target, often for immediate text input. 	<ul style="list-style-type: none"> <code>myEditText.requestFocus()</code>

1.7 Dialog boxes, Menus: Option and Context

Dialog Boxes

Dialogs are small, modal windows that prompt the user to make a decision or enter information. They appear over the current activity and disable background interaction until dismissed.

Types of Dialogs

Android provides several built-in dialog types:

1. **AlertDialog:** The most common type, used for confirmations, alerts, or simple lists. It can have a title, a message, a custom layout, and up to three action buttons (Positive, Negative, Neutral).
1. **Event Handling:** Buttons use `DialogInterface.OnClickListener` to define actions when tapped.
2. **ProgressDialog:** (Deprecated in modern material design; replaced by `ProgressBar` within the UI) Formerly used to show ongoing work.
3. **DatePickerDialog and TimePickerDialog:** Pre-built dialogs with a standard UI for selecting dates and times.

Programmatic Implementation (Kotlin)

Using `AlertDialog.Builder`:

kotlin

```
val builder = AlertDialog.Builder(this)
builder.setTitle("Confirm Action")
builder.setMessage("Are you sure you want to delete this item?")

// Set up the positive (Yes) button
builder.setPositiveButton("Yes") { dialog, which ->
    // Code to execute on "Yes" tap
    Toast.makeText(applicationContext, "Item Deleted", Toast.LENGTH_SHORT).show()
}

// Set up the negative (No) button
builder.setNegativeButton("No") { dialog, which ->
    dialog.dismiss() // Close the dialog
}

// Create and show the dialog
val dialog: AlertDialog = builder.create()
dialog.show()
```

Menus

Menus are used to provide a set of choices or actions in a structured way. Android primarily uses Option Menus and Contextual Menus.

1. Option Menu

The Option Menu (historically the "overflow menu" or App Bar menu) presents primary actions related to the current screen or global application functionality.

- **Location:** Typically located in the Toolbar or AppBar.
- **Trigger:** Displayed when the user taps the three-dot overflow icon or as action icons directly in the App Bar.
- **Implementation:**

- **XML Definition:** You define the menu structure in a resource file (res/menu/main_menu.xml).
- **Activity Overrides:** You override two methods in your Activity:
 - onCreateOptionsMenu(menu: Menu): Inflates the XML menu resource into the Menu object.
 - onOptionsItemSelected(item: MenuItem): Handles click events for specific menu items using their itemId.

Kotlin Implementation Example:

Example:

kotlin

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {  
    // Inflate the menu resource file  
    menuInflater.inflate(R.menu.main_menu, menu)  
    return true  
}  
  
override fun onOptionsItemSelected(item: MenuItem): Boolean {  
    // Handle action bar item clicks here  
    return when (item.itemId) {  
        R.id.action_settings -> {  
            // Handle settings click  
            true  
        }  
        R.id.action_favorite -> {  
            // Handle favorite click  
            true  
        }  
        else -> super.onOptionsItemSelected(item)  
    }  
}
```

2. Context Menu (Contextual Action Mode and Floating Contextual Menus)

Context menus provide actions that relate to a *specific* UI element or item within a list.

A. Floating Contextual Menu (Legacy/Standard)

- **Trigger:** Appears as a floating list when the user performs a **long-press** on a registered View.
- **Implementation:**
 - Register the view for the context menu: registerForContextMenu(myView).
 - Override onCreateContextMenu(menu: Menu?, v: View?, menuInfo: ContextMenuInfo?) to inflate the menu.
 - Override onOptionsItemSelected(item: MenuItem) to handle clicks.

B. Contextual Action Mode (Recommended Material Design Approach)

This is the modern approach, replacing the entire App Bar with a *Contextual Action Bar (CAB)* that displays relevant actions (e.g., Delete, Share) when items are selected (e.g., in a RecyclerView).

- **Trigger:** A long-press usually starts this mode.

- **Implementation:** It uses an `ActionMode.Callback` interface to manage the lifecycle of the CAB.

Feature	Dialog Boxes	Option Menu	Context Menu
Purpose	Alerts, confirmations, prompts	Primary global actions	Actions for specific items/views
Appearance	Modal window over UI	App Bar/Toolbar	Floating list or CAB overlay
Trigger	Code-triggered	Overflow icon/Direct action	Long-press on an element

Parul[®] University | **NAAC** **A++**
Vadodara, Gujarat | GRADE

<https://paruluniversity.ac.in/>

