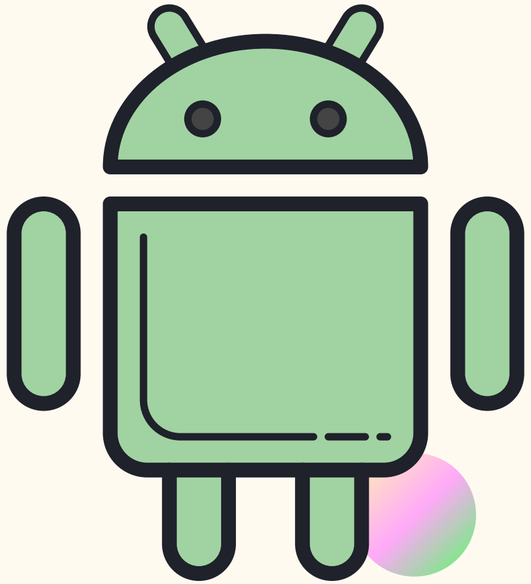
ANDROID

SESSION FIVE



Agenda

- SharedPrefrences
- Android Styling
- Material Design
- Material Design Components

Data Storage

In Android, data storage is a fundamental aspect of app development, enabling applications to save, retrieve, and manage information efficiently.

Storing Data Options

SharedPreferences	Store **private primitive** data in **key-value** pairs
Internal Storage	Storing **private files** that are only accessible by the app.
External Storage	Storing files that need to be **shared** with other apps or **accessed** by the user.
SQLite Database	Storing **structured data** like user profiles, messages, or app content in **private database**.
Network/API Storage	Data stored on **remote servers** and accessed via APIs.

Data Storage

The solution you choose depends on your specific needs ..

1.How much space does your data require?

2.How reliable does data access need to be?

3.What kind of data do you need to store?

4.Should the data be private to your app?

SHARED PREFERENCES

SharedPreferences

- SharedPreferences in Android is a lightweight storage mechanism that allows you to save and retrieve small amounts of data in the form of **key-value pairs.**

- It is commonly used to store user preferences, settings, or other simple **private data** that needs to persist across app sessions.

- We can make use of SharedPreference in situations where we don’t need to store a lot of data and we don’t require any specific structure.

Key Features of SharedPreferences:

- **Key-Value Storage:** Data is stored as key-value pairs, where the key is a string and the value can be a primitive data type (e.g., int, float, boolean, long) ,a String or a set of strings.

- **Persistent Storage:** Data saved in SharedPreferences persists even after the app is closed or the device is restarted.

- **Private to the App:** By default, SharedPreferences files are private to the app and cannot be accessed by other apps.

To get a SharedPreferences object for your application, use one of two methods:

1. getSharedPreferences (name: String, mode: Int): SharedPreferences

What it does:

- Returns a SharedPreferences object that points to a specific file name.
- You can create multiple SharedPreferences files by specifying different names, Name it with the package name of your app unique and easy to associate with the app.
- Use this when you need to store data in multiple SharedPreferences files for different purposes.
- Example: Storing user-specific preferences in one file and app settings in another.

To get a SharedPreferences object for your application, use one of two methods:

1. **getSharedPreferences (name: String, mode: Int): SharedPreferences**

   - Shared Preferences provide **modes** of storing the data (private mode and public mode). It is for **backward compatibility**- use only MODE_PRIVATE to be secure.

   - This method takes two arguments, the first being the name of the SharedPreference(SP) file and the other is the context mode that we want to store our file in.

   - If a preferences file by this name does not exist, it will be created when you retrieve an editor SharedPreferences.edit() and then commit changes Editor.commit().

To get a SharedPreferences object for your application, use one of two methods:

2. getPreferences(int mode)

What it does:

- Returns a SharedPreferences object associated with the current activity.
- The file name is automatically derived from the activity's class name, so you don't need to specify a name.
- Use this when you need to store data that is specific to a single activity.
- Example: Storing UI state or temporary data for a specific activity.

Nested Interfaces of Shared Preferences

- **SharedPreferences.Editor** : Interface used to write(edit) data in the SP file. Once editing has been done, one must commit() or apply() the changes made to the file.

- **SharedPreferences.OnSharedPreferenceChangeListener()** : Called when a shared preference is changed, added, or removed. This may be called even if a preference is set to its existing value. This callback will be run on your main thread.

SharedPreferences.Editor Interface

- Use the **edit()** method of the SharedPreferences object to get an instance of SharedPreferences.Editor :

  SharedPreferences.Editor editor = sharedPreferences.edit();

- Some **Methods** in SharedPreferences.Editor :
  1. remove(String key)
  2. apply()
  3. commit()
  4. putString(String key, String value)
  5. clear()

1. Remove(String key)

- Removes the key-value pair associated with the specified key.
- If the key does not exist, nothing happens.

```
SharedPreferences.Editor editor = sharedPreferences.edit();
editor.remove("username");
```

2. clear()

- Removes all key-value pairs from the SharedPreferences file.
- The file itself is not deleted, but all its contents are cleared.

```
editor.clear();
```

3. **apply()**

- Saves changes asynchronously (in the background).
- Does not block the main (UI) thread.
- Does not return any value.
- Does not guarantee that the changes will immediately persist on the disk.

```
editor.apply();
```

4. commit()

- Saves changes synchronously (blocks the calling thread until the save is complete).
- Returns a boolean indicating whether the save was successful.
- Slower because it blocks the calling thread.
- Safe to use on the main thread, but avoid using it for heavy operations to prevent UI freezes.

```
editor.commit();
```

5. **Writing Data**

The put methods in the Editor interface allow us to save different types of data into SharedPreferences.

The changes must be saved using .apply() or .commit().

- **editor.putInt("age", 22)**
- **editor.putBoolean("isLoggedIn", true)**

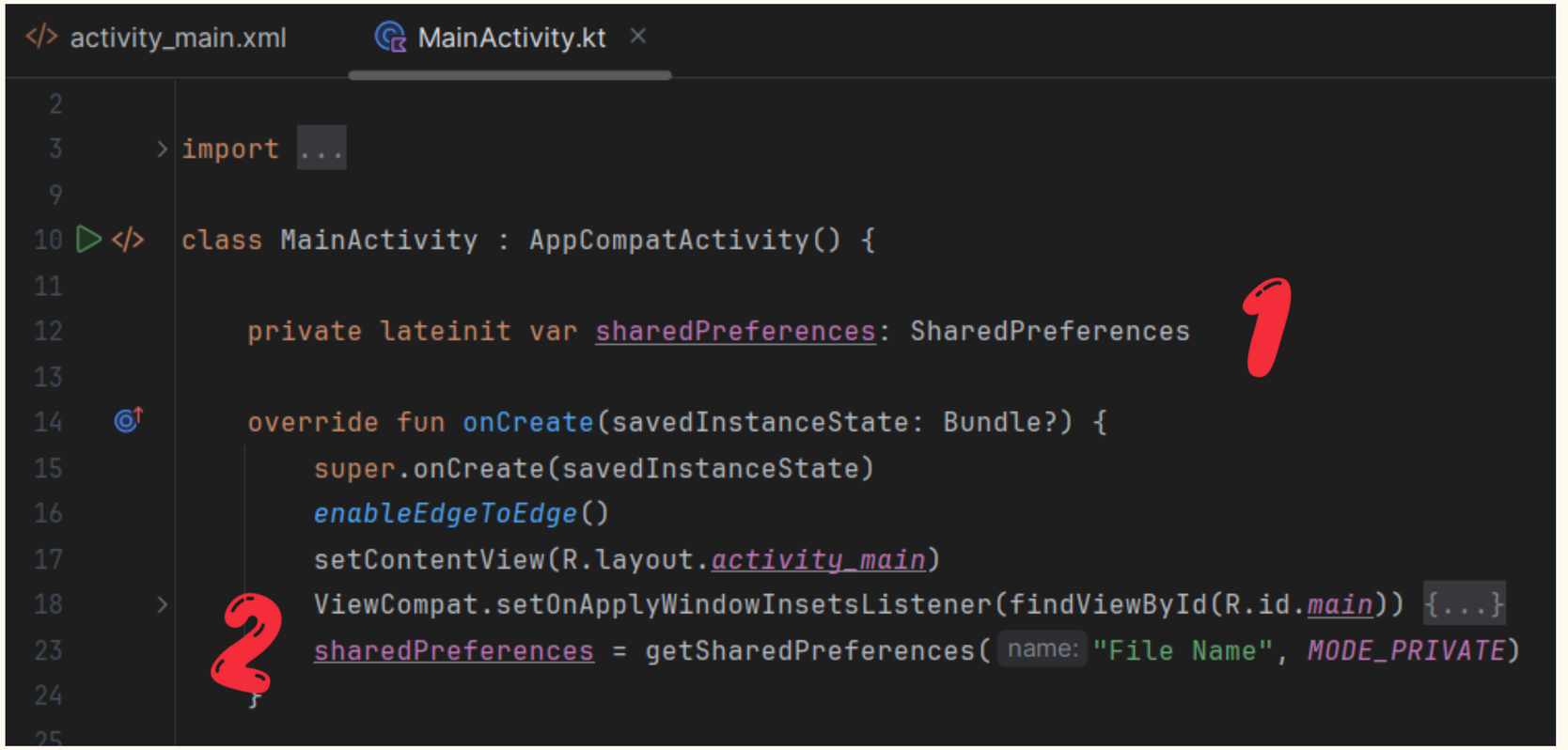
Reading Data (Using get Methods)

- Why No Editor is Needed :
  - The get methods are part of the SharedPreferences object itself, not the Editor.
  - The Editor is only used for writing, updating, or deleting data, not for reading.
  - SharedPreferences is designed to allow direct read access to the stored data without needing an Editor.
- Get methods take two arguments the key, and the default value if the key cannot be found

- **getBoolean(String key, boolean defValue)**
- **getString(String key, String defValue)**

TIME FOR DEMO!

Creating Shared Preferences



Storing The Data :

Store The Data in onPause() or onStop() functions

```
override fun onPause() {
    super.onPause()
    highScore++
    val editor = sharedPreferences.edit()
    editor.putInt("high_score_key",highScore)
    editor.apply()


}
```

Best Practice: Use onPause() for UI-related data and onStop() for background processes

Storing The Data :

Use the scope function “with()” instead of repeatedly writing editor. to improve readability.

```
override fun onPause() {
    super.onPause()
    highScore++
    val editor = sharedPreferences.edit()
    with(editor){
        putInt("high_score_key",highScore)
        apply()
    }
```

- Use the scope function “with()” to perform multiple operations on an object without needing to return it.

We then create a highScore variable, which will be used to display the score in the TextView.

```
class MainActivity : AppCompatActivity() {

    private lateinit var sharedPreferences: SharedPreferences
    private var highScore = 0
    private lateinit var textviewScore : TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        textviewScore =findViewById<TextView>(R.id.test_score)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.activity_main)) {...}
        sharedPreferences = getSharedPreferences( name: "File Name", MODE_PRIVATE)
    }
```

And we use the findViewById function to retrieve the TextView in which we will display the score.

Retrieve The Data :

Retrieve The Data Saved in onCreate() or onStart()

```
    override fun onStart() {
        super.onStart()
        highScore=sharedPreferences.getInt( key: "high_score_key", defValue: 0)
        textviewScore.text = "New Score = $highScore"
    }
}
```

- Use onCreate() for one-time data loading (user settings, static UI elements).
- Use onStart() for frequently updated data (dynamic UI changes, last opened page).

Adding The Clear() Method

```
override fun onPause() {
    super.onPause()
    highScore++
    val editor = sharedPreferences.edit()
    with(editor){
        putInt("high_score_key",highScore)
        clear()
        apply()
    }
```

When combining clear() with put methods , regardless of order when the apply() method is called the clear() method is always called first .

Disadvantages of SharedPreferences

Synchronous (Blocking I/O)

- SharedPreferences operations (especially commit()) run on the main thread, which can cause UI freezes and slow performance.

Stores Data in Plaintext (Unencrypted)

- SharedPreferences saves data in an unencrypted XML file inside internal storage.
- Any rooted device or malicious app with root access can read the stored data.

The Solution...

DataStore

Jetpack DataStore is a modern, asynchronous, and type-safe data storage solution in Android, designed to replace SharedPreferences. It supports Kotlin Coroutines & Flow, ensuring efficient data storage without blocking the UI.

Types of DataStore:

1. **Preferences DataStore** → Stores key-value pairs (like SharedPreferences).
2. **Proto DataStore** → Stores structured data using Protocol Buffers (more efficient & type-safe).

ANDRIOD STYLING

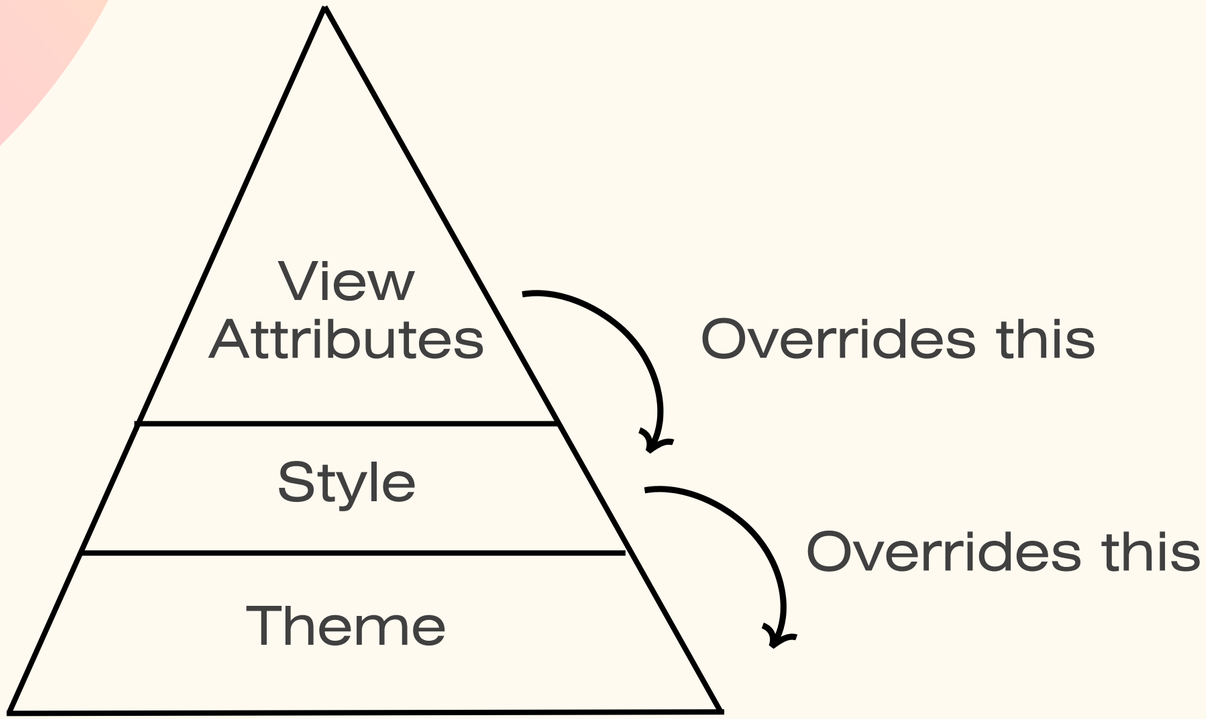
The Android styling system allows developers to define and manage the visual appearance of their apps consistently.

It uses **XML-based resources** to create styles and themes, which can be applied to UI components like buttons, text views, and layouts.

Styles define attributes like colors, fonts, and padding, while **themes** apply these styles globally across the app.

Presedence of each method of styling

In Android, styling can be applied at multiple levels, and the system follows a precedence order to determine which style takes effect when there are conflicts.



Presedence of each method of styling

1. **View Attributes** (Highest Precedence)

   Defined directly in the XML layout using attributes like android:textColor or android:background

Why highest? : These styles are applied directly to the view and override any other styles.

```
<TextView
    android:id="@+id/test_score"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="34sp"
    android:textStyle="bold"
    android:textColor="#FF0000"
```

Presedence of each method of styling

2. View-Specific Styles

- Defined using the style attribute in the XML layout.

activity_main.xml | styles.xml | MainActivity.kt

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="MyCustomTextStyle">MyCustomTextStyle
        <item name="android:textColor">@color/white</item>
        <item name="android:textSize">38sp</item>
        <item name="android:textStyle">bold</item>
    </style>
</resources>
```

1

```xml
<TextView
    android:id="@+id/test_score"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/MyCustomTextStyle"
```

2

Theme

Themes are used to create a consistent look and feel across the application, enhancing user experience and ensuring visual coherence.

- some Components of a Theme :

1. Color Palette
2. Typography
3. Shapes and Borders
4. Spacing and Layout
5. Animations and Transitions
6. Icons and Imagery

Declare a theme

1. In (res/values/themes.xml) file.

```
    <style name="AppTheme" parent="Theme.Material3.DayNight">
        <!-- Custom primary color -->
        <item name="colorPrimary">#6200EE</item>

        <!-- Custom text appearance -->
        <item name="textAppearanceHeadlineMedium">@style/TextAppearance.HeadlineMedium</item>
    </style>

    <!-- Text appearance style -->
    <style name="TextAppearance.HeadlineMedium" parent="TextAppearance.Material3.HeadlineMedium">
        <item name="android:textSize">24sp</item>
        <item name="android:textColor">?attr/colorPrimary</item>
    </style>
</resources>
```

2. Apply the theme to your app in the **AndroidManifest.xml** file:

```
    android:supportsRtl="true"
    android:theme="@style/AppTheme"
    tools:targetApi="31">
```

Presedence of each method of styling

3. **Default Style Attribute in a Theme**
    - Defined in your app's theme **(in res/values/themes.xml)**. It applies to all views of a specific type (e.g., all TextViews or all Buttons) unless you override it.
    - **What it does**: If you define a textViewStyle in your theme, every TextView in your app will use that style by default.
    - **When it’s used**: This is useful when you want to apply a consistent style to all views of a specific type across your app.
    - **Precedence**: It’s overridden by view-specific styles (point 2) and inline styles (point 1).

Presedence of each method of styling

3. Default Style Attribute in a Theme

1

activity_main.xml | styles.xml | MainActivity.kt

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="MyCustomTextStyle">MyCustomTextStyle
        <item name="android:textColor">@color/white</item>
        <item name="android:textSize">38sp</item>
        <item name="android:textStyle">bold</item>
    </style>
</resources>
```

2

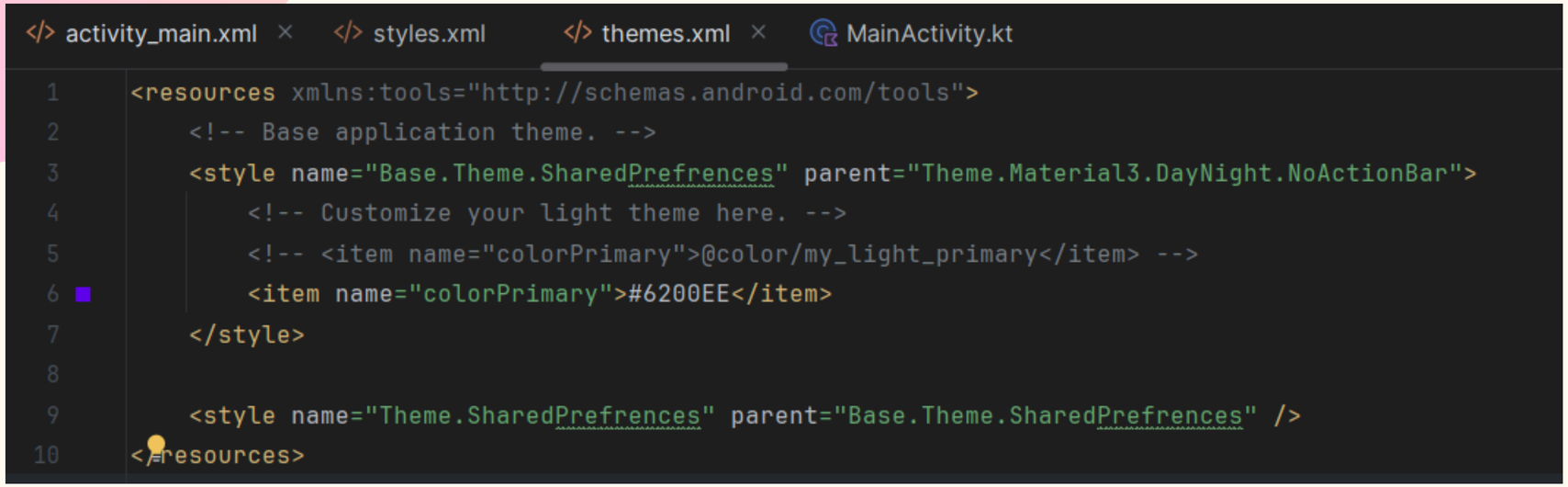
activity_main.xml | styles.xml | themes.xml | MainActivity.kt

```xml
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.SharedPrefrences" parent="Theme.Material3.DayNight.NoActionBar">
        <!-- Customize your light theme here. -->
        <!-- <item name="colorPrimary">@color/my_light_primary</item> -->
        <item name="android:textViewStyle">@style/MyCustomTextStyle</item>
    </style>

    <style name="Theme.SharedPrefrences" parent="Base.Theme.SharedPrefrences" />
</resources>
```

Presedence of each method of styling

4. **Theme Attributes**
   - Theme attributes are reusable values (like colors, fonts, or sizes) defined in your app's theme. They can be referenced in styles or layouts using ?attr/ .



Use Theme Attributes in a layout or style:

- In a layout (res/layout/activity_main.xml):

```
<TextView
    android:id="@+id/textView2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/kotlin"
    android:textColor="?attr/colorPrimary"
```

- In a style (res/values/styles.xml):

activity_main.xml × styles.xml × themes.xml MainActivity.kt

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3      <style name="MyCustomTextStyle">MyCustomTextStyle
4          <item name="android:textColor">?attr/colorPrimary</item>
5          <item name="android:textSize">38sp</item>
6          <item name="android:textStyle">bold</item>
7      </style>
```

Presedence of each method of styling

5. **Default System Theme**

- The default system theme is the base theme provided by Android. If you don’t customize anything, your app will use these default styles.
- Android provides themes like **Theme.Material3.DayNight** or **Theme.AppCompat**.
- These themes include default styles for all UI components (e.g., buttons, text views, etc.).
- In your **AndroidManifest.xml**, you specify the theme for your app:

```
android:supportsRtl="true"
android:theme="@style/Theme.Material3.DayNight"
tools:targetApi="31">
```

Color Resources

By defining colors in the res/values/colors.xml file, you can reuse them throughout your app, making it easier to maintain consistency and update colors globally.

In (res/values/colors.xml) file.

activity_main.xml × colors.xml × MainActivity.kt

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
    <color name="colorPrimary">#6200EE</color>
    <color name="colorPrimaryDark">#3700B3</color>
    <color name="colorAccent">#03DAC6</color>

</resources>
```

Typography

In Android, **sp (scale-independent pixels)** is a unit of measurement used primarily for text sizes in typography. It is similar to dp (density-independent pixels)

- It is used for defining text sizes to ensure that text scales properly based on the user's system-wide font size settings.
- For example, if a user increases the font size in their device settings, text sized with sp will scale accordingly.

RelativeLayout

RelativeLayout is a layout manager that allows you to position child views relative to each other (e.g., place a view below another view) or to the parent layout (e.g., align to top, bottom, start, end).

It is flexible and avoids the need for nested layouts in many cases.

Common Attributes:

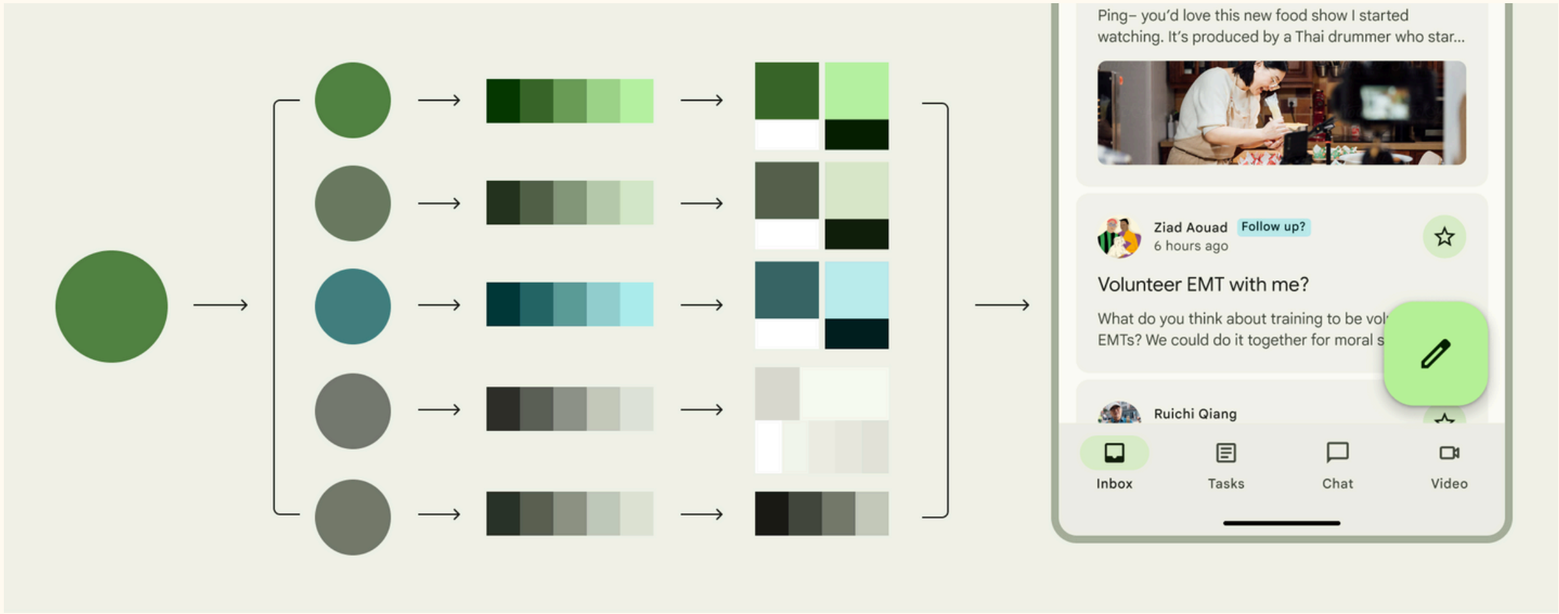
- **android:layout_alignParentTop**: Aligns the view to the top of the parent.
- **android:layout_alignParentBottom**: Aligns the view to the bottom of the parent.
- **android:layout_centerInParent**: Centers the view in the parent.
- **android:layout_toEndOf**: Places the view to the end of another view.
- **android:layout_below**: Places the view below another view.

MATERIAL DESIGN

WHAT IS MATERIAL ?

Material is an adaptable design system of guidelines, components, and tools that support the best practices of user interface design created by Google to help teams build high-quality digital experiences for Android, iOS, Flutter, and the web.

Material Color System

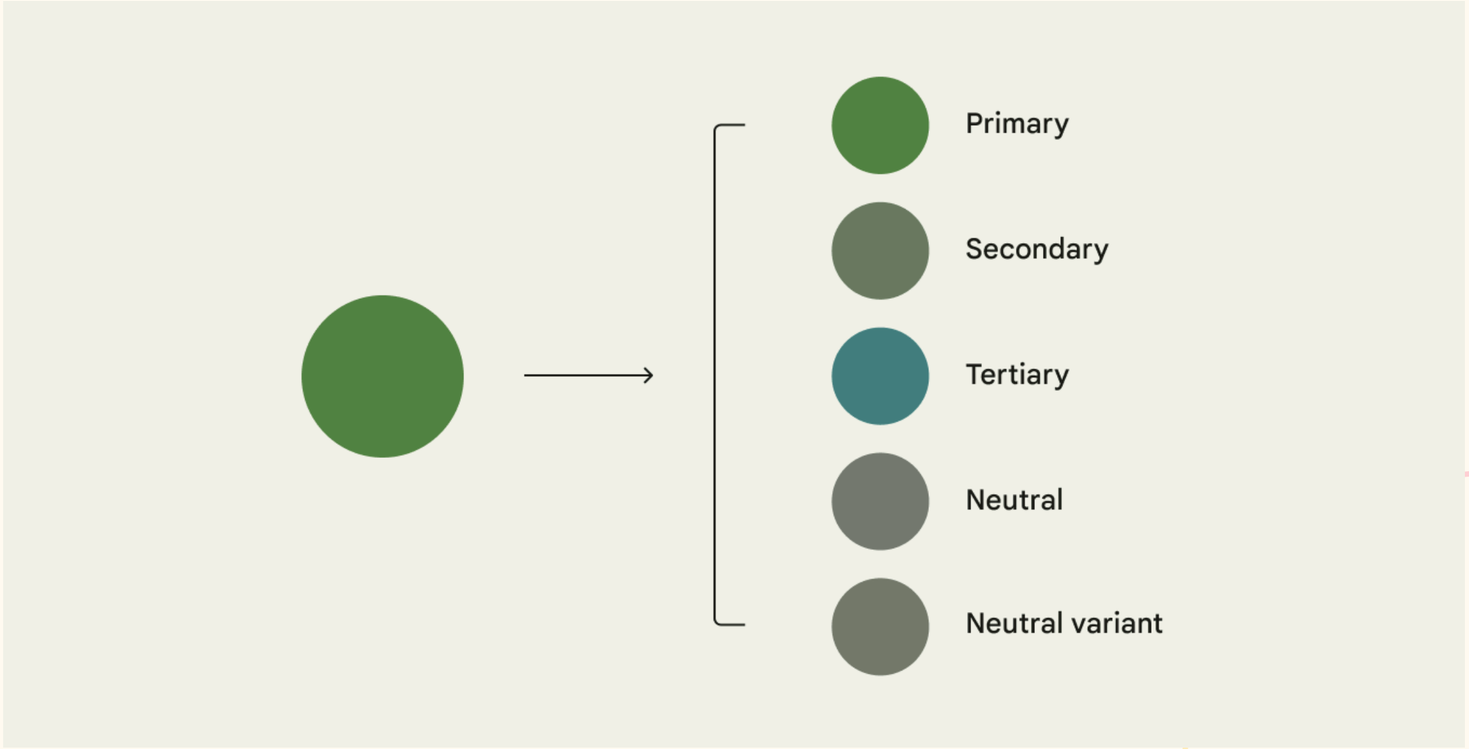


In Material 3, you only need to choose one source color, and the system automatically generates a complete color scheme.

Material Color System

Material's color algorithms manipulate the source color's hue and chroma to generate five complimentary key colors.

1. Primary
2. Secondary
3. Tertiary
4. Neutral
5. Neutral variant



Material Components

are interactive building blocks for creating a user interface.

They can be organized into categories based on their purpose: Action, containment, communication, navigation, selection, and text input.

Components cover a range of interface needs, including:

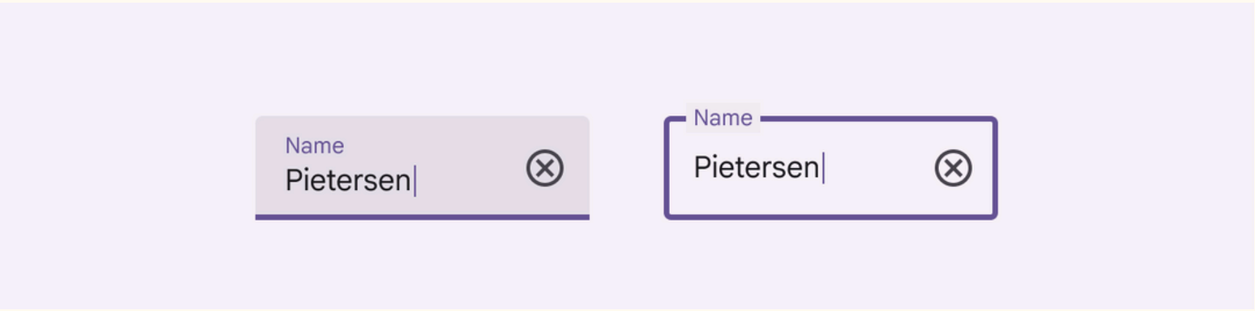
- **Display**: Placing and organizing content using components like cards, lists, and sheets.
- **Navigation**: Allowing users to move through the product using components like navigation_drawers and tabs.
- **Actions**: Allowing users to perform tasks using components such as the floating_action button.
- **Input**: Allowing users to enter information or make selections using components like text fields, chips, and selection controls.
- **Communication**: Alerting users to key information and messages using components such as snackbars, banners, and dialogs.

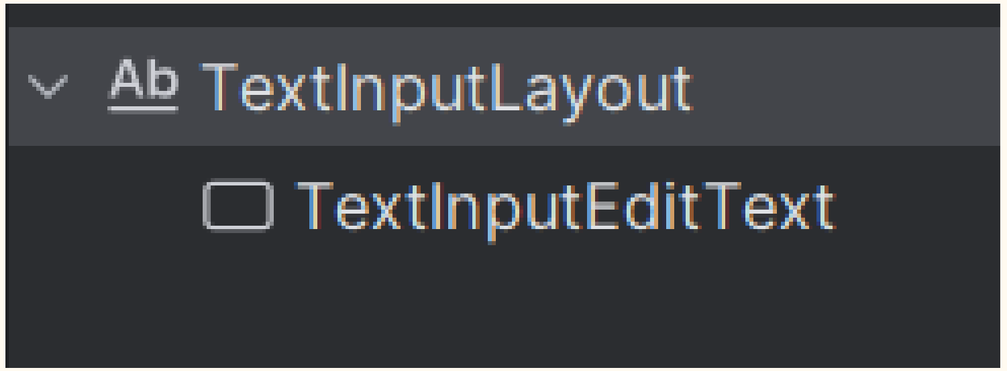
TextField

Composed of TextInputLayout with a child view TextInputEditText

TextInputLayout → Acts as a wrapper that provides additional features like an outline, hint animation, and error handling.

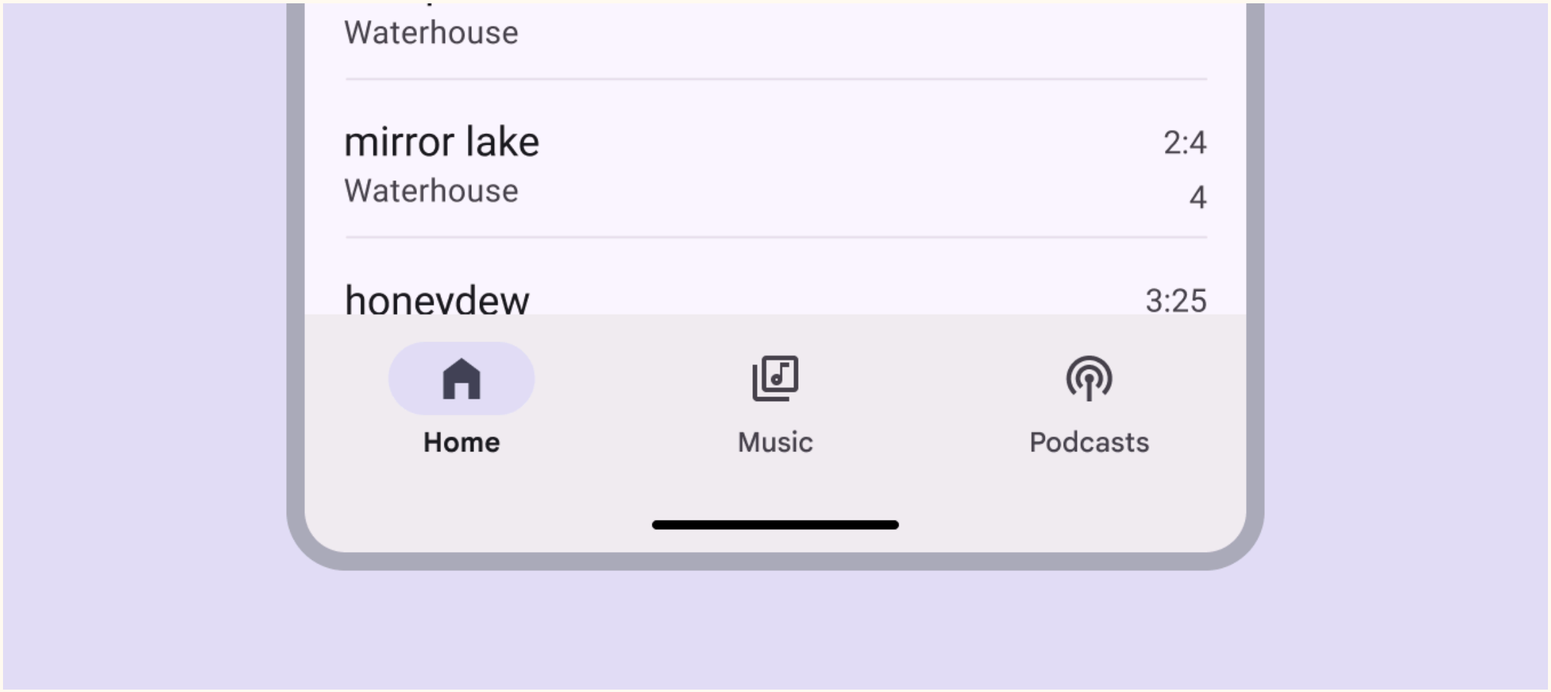
TextInputEditText → The actual input field inside TextInputLayout, where the user types.





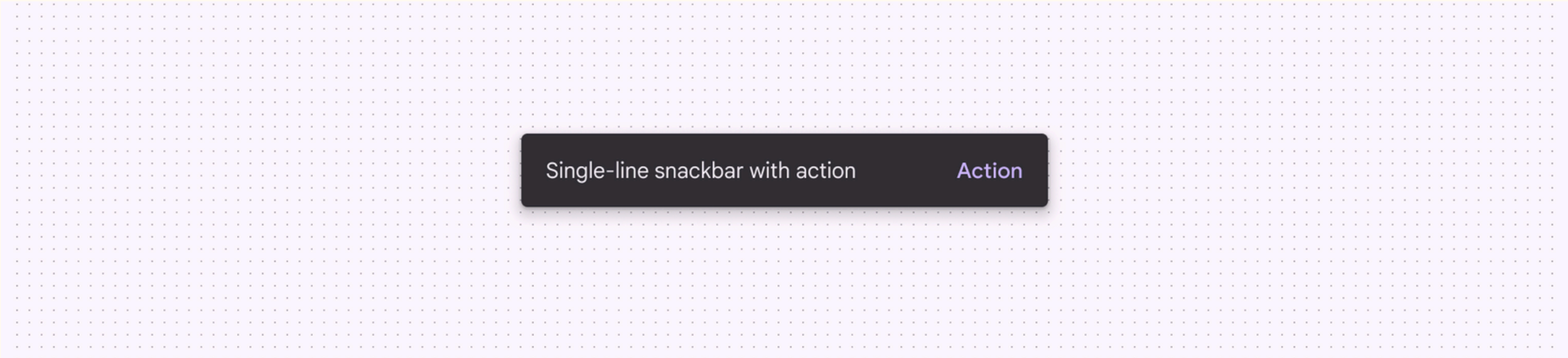
Bottom Navigation

-Allows movement between top level destinations in your app.
-It's an alternate design pattern to a Navigation Drawer.
-Limited to 5 locations max.



Snackbar

-Snackbars shouldn’t interrupt the user’s experience.
-Usually appear at the bottom of the UI.
-Can disappear on their own or remain on screen until the user takes action.



Snackbar Example :

```
val view = findViewById<View>(R.id.activity_main) // The main layout view
Snackbar.make(view, text: "This is a Snackbar", Snackbar.LENGTH_SHORT).show()
```

Add an action to the Snackbar :

2

```
Snackbar.make(view, text: "This is a Snackbar", Snackbar.LENGTH_SHORT)
    .setAction(R.string.action_text){
        //Responds to click on the action
        Snackbar.make(view, text: "Action clicked!", Snackbar.LENGTH_SHORT).show()
    }
    .show()
```

1

activity_main.xml | MainActivity.kt | strings.xml

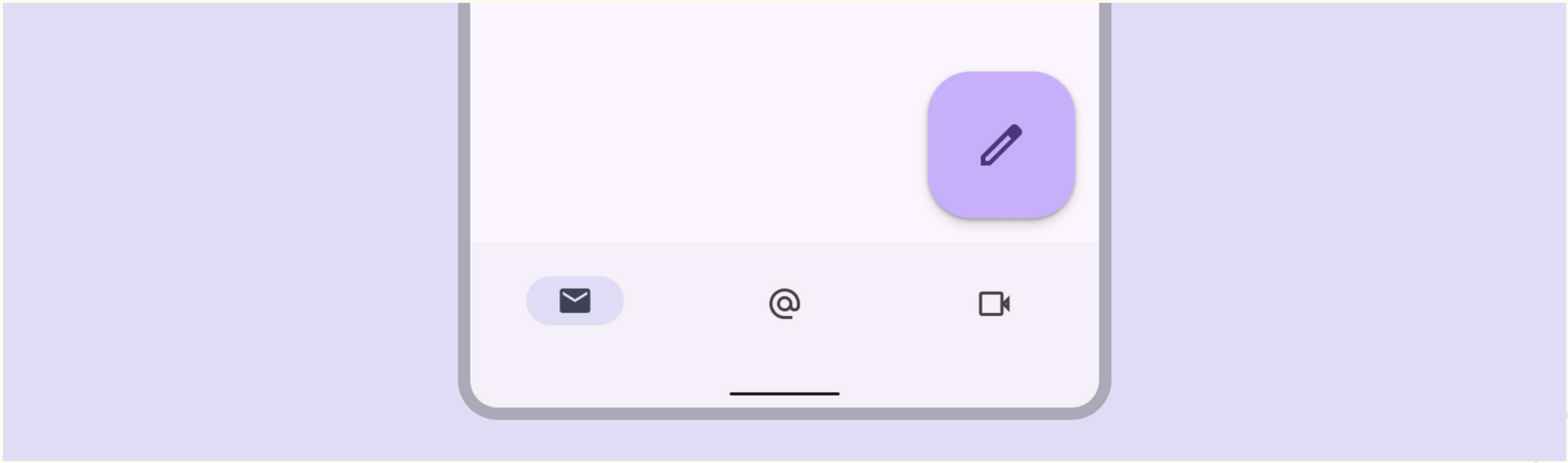
```
1  <resources>
2      <string name="app_name">sharedPrefrences</string>
3      <string name="action_text">Undo</string> />
4  </resources>
```

Floating action buttons

-Use a FAB for the most common or important action on a screen

-Make sure the icon in a FAB is clear and understandable

-FABs persist on the screen when content is scrolling



Floating action buttons

```xml
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/floatingActionButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="16dp"
    android:layout_marginBottom="16dp"
    android:clickable="true"
    android:focusable="true"
    app:fabSize="normal"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:srcCompat="@android:drawable/btn_star_big_on" />
```

1

2

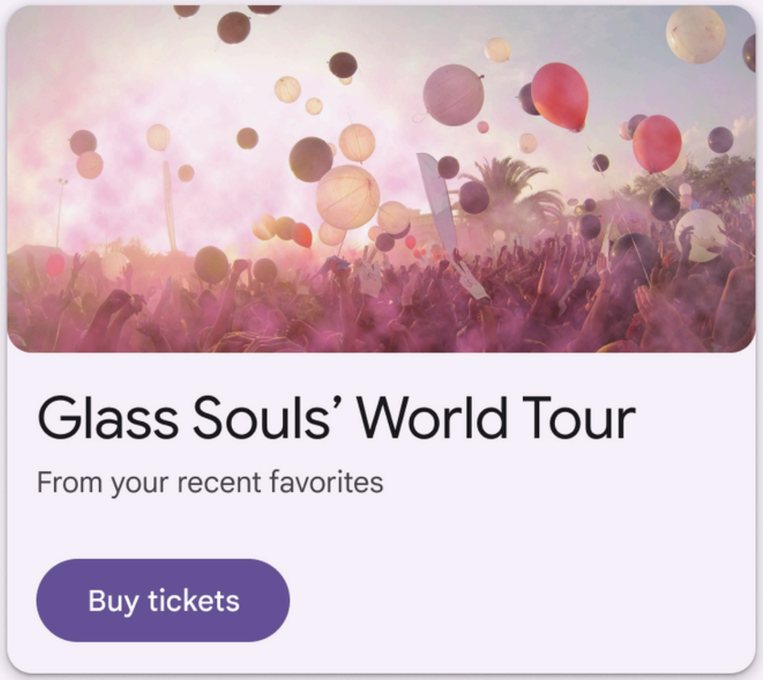
```kotlin
val button: FloatingActionButton = findViewById(R.id.floatingActionButton) // Replace with your button ID
button.setOnClickListener {
    Toast.makeText( context: this@MainActivity,  text: "Click Home Fab Button", Toast.LENGTH_SHORT).show()
}
```

Cards

-Use cards to contain related elements

-Contents can include anything from images to headlines, supporting text, buttons, and lists

-Use MaterialCardView



Cards

```
<com.google.android.material.card.MaterialCardView
    android:id="@+id/materialCardView"
    android:layout_width="0dp"
    android:layout_height="127dp"
    android:layout_margin="16dp"
    android:layout_marginStart="16dp"
    android:layout_marginBottom="352dp"
    app:cardCornerRadius="12dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginStart="16dp"
        android:layout_marginBottom="192dp"
        android:orientation="vertical"
        android:padding="16dp">

        <!-- Title Text -->
        <TextView...>

        <!-- Description Text -->
        <TextView...>
    </LinearLayout>
</com.google.android.material.card.MaterialCardView>
```

Thank You