Android Reference

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

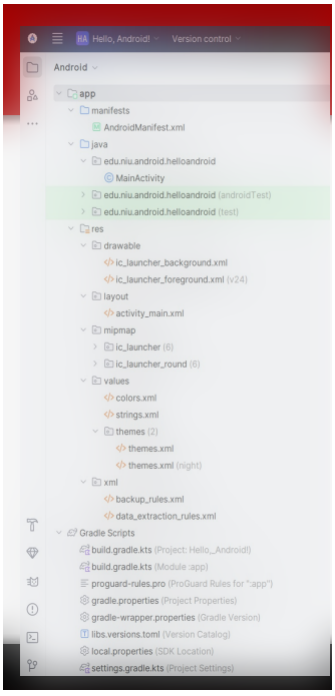
Contents

The Basics

- **Layout file for apps "look"**: Note that the XML layout file for our app's "look" is named activity_main.xml
- **Creating a new project**: Click on the Empty Views Activity template to highlight it and click Next

  1. Enter the name of the project
  2. Enter the package name as edu.niu.zid.projectname
  3. Set language to Java
  4. Choose API 22 ("Lollipop"; Android 5.1) for the minimum SDK
  5. Click finish

- **Directory structure**: please be sure that you have no additional layers than what you see here.



Notice at the highest level in the hierarchy of our project folders that there are two main "pieces" to the Android project:

1. app
2. Gradle Scripts

Within the app, there are really three folders we are going to be concerned with for now:

1. manifests
2. java
3. res

- **Manifests**: The directory named manifests holds one xml file - so far. It is the AndroidManifest.xml file.
- **Java (directory):** The directory named java holds one Java source files - so far. It is the MainActivity.java file
- **Res (directory)**: The directory named res holds resource files, such as utility xml files for defining strings, themes, menus, layouts, dimensions, images, sounds, etc
- **MainActivity.java**: This is the logic/controller file. It's the Java class that defines what your app does when it runs. It extends AppCompatActivity (or another activity class), and inside it you set up event handling, lifecycle methods (like onCreate), and the code that reacts to user interactions. For example, if a button is clicked, the code for what happens lives here.

  The simplest MainActivity.java file is as follows

```
0  package YOURPACKAGENAME
1
2  import androidx.appcompat.app.AppCompatActivity;
3
4  import android.os.Bundle;
5
6  public class MainActivity extends AppCompatActivity {
7
8      @Override
9      protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_main);
12     }
13 }
```

  In this code, the most important import statement is the import for class android.appcompat.app.AppCompatActivity

  MainActivity.java class extends, or inherits from, superclass AppCompatActivity class found in Android API package androidx.appcompat.app.AppCompatActivity

  AppCompatActivity itself is a subclass itself of Activity.

  An Activity provides the screen with which users can interact, in other words, the User Interface, or UI.

  The MainActivity class is meant to be used as the Controller for every app we develop

  Method onCreate() of an Android app is called automatically upon launch just like the main() method is called automatically by the Java Virtual Machine, or JVM, upon launch of a Java applications.

  Inside method onCreate(), we create the initial View for the app, controlled by this Activity.

In method onCreate(), the superclass' onCreate() method is first called.

Then the View, or opening screen, for this Activity is set by calling method setContentView().

Method setContentView() is inherited from the AppCompatActivity , or Activity class.

Its API is

```
void setContentView(int layoutResID)
```

layoutResID is a resource found in the subfolder named layout in the folder named res.

This ID defaults to: *R.layout.activity_main.*

In MainActivity.java, we refer to and access activity_main.xml using the syntax

```
R.layout.activity_main
```

activity_main is a static constant of the static final class layout defined inside the R.java class or file.

The R.java file was automatically created when we began our project

R stands for resources and "represents" the res directory.

- **activity_main.xml**: This is the UI/layout file. It defines the visual structure of your screen using XML: buttons, text fields, images, layouts, etc. You don't put behavior here—only the arrangement and styling of the interface elements.

  activity_main is an xml file, a resource located in the layout subdirectory of the res directory

  It was also automatically created when we began our project using the template we chose.

  We can think of XML as a markup language similar to HTML, but with user-defined tags

- **Intro to design editor**: With the activity_main.xml file focused, click on this small button near the upper right

Note that this opens the Design editor in which we can drag and drop widgets into our app. More on this to come!

- **Intro to XML syntax**: In XML, an non-empty element uses the general syntax shown here

  ```
  <tagName attribute1="value1">Element Content</tagName>
  ```

  For example:

  ```
  <color name="red">#FFFF0000</color>
  ```

  which can be interpreted as there is an element color that has the name red and corresponds to the RGB value #FFFF0000.

  If an element has no content yet or will never have any content, we can use this syntax

  ```
  <tagName attribute1="value1" attribute2="value2".../>
  ```

- **XML Naming Rules**: XML elements must follow these naming rules:

  1. Names can contain letters, numbers, and other characters.
  2. Names cannot start with a number or punctuation character.
  3. Names cannot start with the letters xml (or XML, or Xml, etc).
  4. Names cannot contain spaces.

- **XML Comments**: Documentation in XML is done with

  ```
  <!-- comment text -->
  ```

- **Simple activity_main.xml Example, TextView and ConstraintLayout**: Let's consider a simple XML example

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/app_name"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

There are two elements in our current activity_main.xml:

1. ConstraintLayout
2. TextView

The TextView element is nested inside the ConstraintLayout element

A ConstraintLayout allows us to position and size elements in a flexible way.

One way is relative positioning, or where an element is placed on the screen, or view, relative to another.

The ConstraintLayout element has several attributes.

The android:layout_width and android:layout_height attributes define the size of the ConstraintLayout.

Their value match_parent means that it will be as big as its parent element, which is the screen of the Android device in this case.

The TextView element is empty, or, in other words, has no content, and has, by default, seven attributes.

The first three are layout_width, layout_height and text.

The android:text attribute specifies the content, or the words, of the TextView to be shown on the screen of the Android device.

By default, the four attributes that follow layout_width, layout_height and text are for positioning of the TextView within the ConstraintLayout that will be discussed later

A TextView element is an instance of the TextView class, which puts a label on the screen.

The android:layout_width and android:layout_height attributes define the size of the TextView.

The value wrap_content means that it will be as small as possible so that their contents (the text or words) fit inside it.

In other words, the element "wraps" around its content.

By default, the TextView appears in the middle of the screen

The four app:layout attributes specify the relative position of the TextView element's four sides

In this example, they are relative to the "parent", or the ConstraintLayout in which the Textview is nested

- **strings.xml**: is the resource file for text values in your app. Stores text like titles, labels, button names, or messages as key–value pairs. keep all text centralized for easy updates, consistency, and support for localization

  You reference them in code (getString(R.string.welcome_message)) or XML (@string/welcome_message).

  One string is defined in the strings.xml file by default

```
<resources>
    <string name="app_name">Hello, Android!</string>
<resources>
```

  The syntax for defining a string is

```
<string name="stringName">valueOfString</string>
```

- **themes.xml**: defines the overall visual style of your Android app.

  Central place to configure app-wide appearance, like colors, fonts, backgrounds, status bar style, and widget looks.

  A theme is a collection of style attributes applied globally.

- **themes.xml example**:

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->

    <style name="Base.Theme.MyApplication"
    ↪ parent="Theme.Material3.DayNight.NoActionBar">
        <!-- Customize your light theme here. -->
        <!-- <item name="colorPrimary">@color/my_light_prim
        ↪ ary</item> -->
    </style>

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

It is used to define styles that the app uses.

We can modify a theme by adding an item element using this syntax

```
<item name="styleAttribute">valueOfItem</item>
```

The name of the theme attribute that specifies the text size inside a TextView is android:textSize.

Let us change the default text size to 40 by adding the following code to the themes.xml file just above the line </style>

```
<item name="android:textSize">40sp</item>
```

- **Removing and adding action bar (themes.xml)**: Also, this line removes the commonly shown Action Bar:

```
<style name="Base.Theme.HelloAndroid"
↪ parent="Theme.Material3.DayNight.NoActionBar">
```

Note: The Empty Views Activity template does not provide a default Action Bar

If we change the line to

```
<style name="Base.Theme.HelloAndroid"
↪ parent="Theme.AppCompat.Light.DarkActionBar">
```

And make two other changes, we will see the Action Bar with the name of our app displayed

- **colors.xml**: The colors.xml file in the res > values directory is another automatically generated XML file.

It is used to define colors we want to use in our apps

The syntax for defining a color is

```
<color name="colorName">valueOfColor</color>
```

The color value is defined using hexadecimal, or base 16, notation

#FFrrggbb (uses RGB color system) where:

- rr = amount of red in color
- gg = amount of green in color
- bb = amount of blue in color

- **colors.xml example**: If we create the color in colors.xml

```
<color name="colorPrimary"> #FF3F51B5 </color>
```

Then, in themes.xml, we can add or uncomment the line

```
<item name="colorPrimary">@color/colorPrimary</item>
```

inside of the action bar tag

```
<style name="Base.Theme.Test3"
    parent="Theme.AppCompat.Light.DarkActionBar">
    <item name="colorPrimary">@color/colorPrimary</item>
</style>
```

Now we have a blue action bar.

- **AndroidManifest.xml**: The AndroidManifest.xml file is located in the manifests directory.

  It specifies the resources that the app uses, such as activities, the file system, the Internet, and hardware resources.

  Before a user downloads an app on Google Play, the user is notified about these details

- **Changing the launcher icon**: When your app is installed on a device, its icon and name appear with all other installed apps in the launcher

  Users can use the icon to launch our app on their device.

  We should always supply a launcher icon for our app

A launcher icon for a mobile device should be 48 x 48 dp.

Because various devices can have different screen densities, we can supply several launcher icons, one for each density

If we provide these different versions of our icon, we need to follow the 2/3/4/6/8 scaling ratios between the various densities from medium (2) to xxx-high (8)

If we supply only one icon - as we will do here - Android Studio will use that icon and expand its density as necessary.

If we plan to publish our app, we should provide a 512 x 512 launcher icon for display in Google's app store.

Right mouse click on mipmap in the project structure along the right side and choose New and then Image Asset.

To set the launch icon for the app to hi.png, we assign the String @mipmap/hi to the android:icon attribute and the String @mipmap/hi_round to the android:roundIcon attribute of the application element in the AndroidManifest.xml file.

The @mipmap/hi expression defines the resource in the mipmap directory (of the res directory) whose name is hi (note that we do not include the extension) and the same for hi_round

```
android:icon="@mipmap/hi"
↪ android:roundIcon="@mipmap/hi_round"
```

- **Orientation**: Sometimes, we want the app to run in only one orientation

  In other words, we do not want the app to rotate when the user rotates his or her device.

  Inside the activity element (in the manifest), we can add the android:screenOrientation attribute and specify either portrait or landscape as its value

  For example, if we want our app to run in vertical orientation only, we add

  ```
  android:screenOrientation="portrait"
  ```

  Note that we can control the behavior of the app on a per activity basis

- **Gradle build system**: Android Application Package, or APK, is the file format for distributing applications that run on the Android operating system.

  The file extension is .apk

To create an apk file, the project is compiled and its various parts are packaged into the apk file.

The apk file can be found in the project directory:

projectName/app/build/outputs/apk

These .apk files are built using the gradle build system, which is integrated in the Android Studio environment.

When we start creating an app, the gradle build scripts are automatically created

They can be modified to build apps that require custom building

- **Debugging**: Just like in a regular Java program, we can send output to the console in addition to displaying data on the screen.

  For this, we can use one of the static methods of the Log class.

  The Log class is part of the android.util package

  Selected methods of the Log class include d, e, i, v, w.

  They all have the same parameter list and return type; for example, the API of d is

```
0  public static void d(String tag, String message)
```

  - **Log.d(String tag, String msg):** Debug: Used for debugging messages. These are usually filtered out in release builds.
  - **Log.e(String tag, String msg):** Error: Used to report error conditions. This is the highest-severity logging method.
  - **Log.i(String tag, String msg):** Info: Used for general information messages that highlight the progress of the application.
  - **Log.v(String tag, String msg):** Verbose: Used for the most detailed log messages, often too much for normal use, but helpful during deep debugging.
  - **Log.w(String tag, String msg):** Warning: Used to report potential issues or unexpected states that aren't necessarily errors.

  tag identifies the source of the message and can be associated with a "filter" (described in a few slides).

  message is the String to be output.

  To run the app in debug mode, we click on the debug icon on the toolbar.

  The app runs and stops at the first breakpoint.

  The Debug tab will open in the panel at the bottom of the screen.

Here we will see some debugging information and tools.

Under Frames, we can see where in the code we are currently executing.

To resume the app, we click on the green Resume icon at the top left of the panel.

As we resume, stop at breakpoints, and resume the app a few times, the values of the various variables in our app are displayed under Variables.

Under Variables, we can check the values of the various variables

If the app is running on a device, we can still log output statements in Logcat

This is much faster than starting the emulator.

- **Logcat**: Output from logging statements show up in the Logcat

  To open the Logcat, click on the Logcat tab at the bottom of the IDE.

  We can filter the output listed in the Logcat by clicking in the dropdown in the upper right of the Logcat

  Suppose we create a new filter named *f_mainactivity*, with a tag MainActivity.

  Now that a filter has been created along with its tag, we can output messages to Logcat.

  Add the following line to the onCreate method of MainActivity.java but after the call to super.onCreate

```
0  Log.w("MainActivity", "Inside onCreate!");
```

  The output we will see in Logcat is: Inside onCreate!

  Or another way to do it

```
0  ...
1  public static String MA = "MainActivity";
2  ...
3  setContentView(R.layout.activity_main);
4  Log.w(MA, "View resource: " +
5  R.layout.activity_main);
6  ...
```

Reference

2.1 Includes

- androidx.appcompat.app.AppCompatActivity;
- android.os.Bundle;
- android.util.Log;
- android.view.View;
- android.view.Gravity;
- android.graphics.Color;
- android.widget.EditText;
- android.widget.TextView;
- java.text.NumberFormat;
- android.text.TextWatcher:
- android.text.Editable
- java.lang.CharSequence
- android.graphics.Point;
- android.widget.Button;
- android.widget.GridLayout;
- androidx.constraintlayout.widget.ConstraintLayout;
- androidx.constraintlayout.widget.ConstraintSet;
- androidx.constraintlayout.widget.Guideline
- import androidx.constraintlayout.widget.Barrier;
- android.view.ViewGroup;
- android.content.Context;
- android.content.DialogInterface;
- androidx.appcompat.app.AlertDialog;
- android.graphics.Typeface;
- android.view.Display;
- android.widget.RelativeLayout

2.2 Important information

- **AVD (Android virtual device)**: It's basically an emulated device configuration that runs inside the Android Emulator. An AVD defines things like:
  - **Device model**: (screen size, resolution, density, RAM, etc.)
  - **System image**: (Android version, API level, Google Play support, etc.)
  - **Hardware features**: (camera, GPS, sensors, etc.)

  So when you launch an emulator in Android Studio, you're really starting up an AVD that behaves like a specific Android phone or tablet.

- **Screen Density**: screen density means how many pixels are packed into a physical area of the screen, usually measured as dots per inch (dpi).

  A screen with high density has many pixels in a small space, making things look sharper but also smaller if not scaled.

  A screen with low density has fewer pixels in the same area, so things look larger but less sharp.

  Android groups devices into "density buckets" so developers don't have to manually calculate for every possible screen:

  - **ldpi (low)**: ∼120 dpi
  - **mdpi (medium)**: ∼160 dpi (baseline)
  - **hdpi (high)**: ∼240 dpi
  - **xhdpi (extra-high)**: ∼320 dpi
  - **xxhdpi (extra extra high)**: ∼480 dpi
  - **xxxhdpi (extra extra extra high)**: ∼640 dpi

  Android uses this formula under the hood:

  $$px = dp \times \frac{dpi}{160}$$

  Where

  - dp = your value in density-independent pixels (e.g., 20)
  - dpi = the device's actual screen density in dots-per-inch
  - 160 = the baseline density (mdpi)

  Even though the number of pixels changes, the physical size (in inches or mm) stays about the same. That's because on a denser screen, pixels are smaller, so Android uses more of them to maintain the same real-world size.

  So your 20dp button will look like the same size button whether it's on a cheap low-res phone or a modern super high-res one.

- **Layout Params**: Generally, a layout class like ConstraintLayout or RelativeLayout contains a static inner class that contains XML attributes that allow us to arrange the components within the layout. This static inner class is often named **LayoutParams**

Every child view inside a ViewGroup needs layout parameters (LayoutParams).

These tell the parent how big the child should be and how it should be positioned.

Different ViewGroups define their own rules:

  - **LinearLayout.LayoutParams**: weight, gravity
  - **FrameLayout.LayoutParams**: gravity
  - **ConstraintLayout.LayoutParams**: constraints like topToBottom, leftToLeft, etc.

- **Positioning components inside ConstraintLayout**: We will use the attributes of ConstraintLayout.LayoutParams to position the XML elements on the screen
- **Ids**: The android:id attribute allows us to give an id to an XML element

  The syntax for assigning an id to an XML element is

```
android:id = "@+id/idValue"
```

- **Default colors (defined in @android:color)**:
  - **@android:color/black**: #FF000000
  - **@android:color/darker_gray**: #FF444444
  - **@android:color/dim_gray**: #FF696969
  - **@android:color/gray**: #FF888888
  - **@android:color/light_gray**: #FFCCCCCC
  - **@android:color/white**: #FFFFFFFF
  - **@android:color/red**: #FFFF0000
  - **@android:color/green**: #FF00FF00
  - **@android:color/blue**: #FF0000FF
  - **@android:color/yellow**: #FFFFFF00
  - **@android:color/cyan**: #FF00FFFF
  - **@android:color/magenta**: #FFFF00FF
  - **@android:color/transparent**: #00000000 (fully transparent)

  We can get access to these default colors in java using

```
int color = context.getResources().getColor(android.R.color.
↪ colorname)
```

- **ConstraintLayout Enable Autoconnection to Parent**: When you enable it, every new view you drag into the ConstraintLayout will automatically get constraints to its closest edges of the parent (top, bottom, start, end).
- **Styles vs Themes**: A style relates to a UI component or a View. A theme relates to an activity; it can even relate to the whole app.

We can also ”theme” an app with a style by editing the AndroidManifest.xml file, changing the android:theme attribute of its application element using this syntax:

```
android:theme="@style/nameOfStyle"
```

- **findViewById**: findViewById is a fundamental method in Android development used to obtain a reference to a View object defined in your XML layout files. This method allows you to interact with UI elements programmatically, such as setting text, handling clicks, or changing visibility.

  For example,

```
EditText billEditText =
    (EditText)findViewById(R.id.amount_bill);
```

  findViewById returns the View that is part of the layout xml file that was inflated in method onCreate() of the Activity class.

  If we expect to retrieve a TextView and we want to assign the View retrieved to a TextView, we need to cast the View returned by method findViewById() to a TextView:

- **View Event handling with XML (only onClick)**: To set up a click event on a View, we use the android:onClick attribute of the View and assign to it a method using the form:

```
android:onClick="methodName"
```

  The event will be handled inside that method, which must have the following API

```
public void methodName(View v)
```

  v is the View where the event happened.

  When the user clicks on the button, we execute inside calculate, and its View parameter is the Button; if we have the following statement inside calculate

```
Log.w("MainActivity", "v = " + v);
```

  Inside LogCat, we have something like:

```
v=android.widget.Button@425a2e60
```

The value above identifies the Button

- **IME**: It's the software keyboard or other input system (like voice typing, handwriting, predictive text) that allows users to enter text into your app.

  Every EditText communicates with the IME through the Input Method Framework (IMF) in Android.

  When you tap into an EditText, the IME pops up (usually the on-screen keyboard).

- **Get size of screen**:

```
import android.graphics.Point;

Point size = new Point();
getWindowManager().getDefaultDisplay().getSize(size);
```

- **Create grid layout in java (controller)**:

```
GridLayout gridLayout = new GridLayout(this);
gridLayout.setRowCount(int m);
gridLayout.setColumnCount(int n);
```

- **Adding views to GridLayout**:

```
gridLayout.addView(view, w, h);
```

- **setContentView**: the method setContentView(...) is used inside an Activity to specify which layout file (XML) should be used as the UI for that screen.

  Calling `setContentView(R.layout.some_layout)` tells the activity:

  "Use the layout resource some_layout.xml from the *res/layout* folder as the UI for this screen."

  If you're using programmatic UI (creating Views in Java instead of XML), you can pass a View object directly,

- **ViewGroup**: In Android, everything you see on screen is either a View or a ViewGroup.

  A View is a basic UI element — e.g. Button, TextView, EditText, ImageView.

  A ViewGroup is a container that holds other Views (and even other ViewGroups).

So, a ViewGroup is essentially a layout manager. It defines how child views are positioned, sized, and arranged inside it.

- **LinearLayout**: places children in a row or column.
- **RelativeLayout (older, mostly replaced by ConstraintLayout)**: places children relative to each other.
- **ConstraintLayout**: a flexible, modern layout system.
- **FrameLayout**: stacks children on top of each other.
- **RecyclerView**: a powerful scrolling container for lists/grids.

Note: A ViewGroup is a subclass of View.

- **Giving id to view in java**: We can use

```
View.generateViewId()
```

as an argument to `view.setId()`. The method `generateViewId()` is a static method in the View class. It creates a unique integer ID at runtime that's guaranteed not to collide with other view IDs.

- **The finish() method**: finish() is a method of the Activity class. It tells Android:

  > "I'm done with this Activity, please close it and remove it from the back stack."

  So if you call finish() inside an Activity, that activity will end and control will go back to the previous one (or exit the app if it was the only activity).

- **Converting px to dp in java**: In Android, px (pixels) and dp (density-independent pixels) are not the same thing. You almost always want to work in dp because it scales properly across different screen densities.

```
float dp = px /
    ↪ context.getResources().getDisplayMetrics().density; // px
    ↪ to dp

float px = dp *
    ↪ context.getResources().getDisplayMetrics().density; // dp
    ↪ to px
```

when a Java function in Android Studio takes an int representing a dimension (like width, height, margin, radius, stroke width, etc.), it is almost always in raw pixels (px) — not dp.

`context.getResources().getDisplayMetrics().density` gets the screens density factor (dpi / 160)

- **xmlns:android="http://schemas.android.com/apk/res/android"**: One of the most important pieces of XML in Android layouts.

  It tells the XML parser: "Whenever you see an attribute that starts with android:, it belongs to the Android system's XML namespace, located at this URI."

You must include this in:

- Any layout XML (e.g. activity_main.xml)
- Any drawable XML, menu XML, or style XML that uses android: attributes

Basically, any file where you use attributes like android:layout_width, android:padding, android:text, etc.

you only need to declare that line once, and it always goes on the root element of your XML layout (like <ConstraintLayout>, <RelativeLayout>, <LinearLayout>, etc.).

- **The dp function (java)**: In Android, UI dimensions shouldn't be defined in plain pixels because screens have different pixel densities (dpi). Instead, we must use dp (density-independent pixels), which scale consistently on all screens.

  XML converts dp to px for us, but in java we must do it ourselves. So, we write the following function

```
private int dp(int value) {
    float density =
    ↪  getResources().getDisplayMetrics().density;
    return (int) (value * density);
}
```

  We pass in a **dp** values, and it gets converted to px.

- **Animation directory**: In order to create and use animations in our apps, we need to create a **Android Resource Directory** in the **res** directory with resource type **anim**. Then, we can define animations in XML (one animation per XML file).

- **Action bar**

```
<resources>
  <!-- Base application theme. -->
  <style name="Theme.MusicStore"
  ↪  parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="android:textSize">15sp</item>
    <item name="colorPrimary">@color/colorPrimary</item>
  </style>

</resources>
```

- **No action bar**

```
<resources xmlns:tools="http://schemas.android.com/tools">
    <!-- Base application theme. -->
    <style name="Base.Theme.MusicStore"
    ↪ parent="Theme.Material3.DayNight.NoActionBar">
        <!-- Customize your light theme here. -->
        <!-- <item name="colorPrimary">@color/my_light_prima
        ↪ ry</item> -->
    </style>

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

- **Menus**: Menus in Android Studio (and Android apps generally) are UI components that let users interact with common actions — similar to menus in desktop apps, but adapted for mobile.

  A menu is a set of action items your app displays in:

  - The top App Bar (toolbar)
  - A vertical overflow menu (three-dot icon)
  - A contextual action bar
  - Popup menus inside a view

  Menus are defined in `res/menu`

- **Builtin icons**: These are the most frequently used system drawables (they all live under android.R.drawable):

  - **@android:drawable/ic_menu_preferences**: Settings or preferences
  - **@android:drawable/ic_menu_edit**: Edit / modify item
  - **@android:drawable/ic_menu_attach**: Attach file
  - **@android:drawable/ic_menu_send**: Send / share
  - **@android:drawable/ic_menu_delete**: Delete item
  - **@android:drawable/ic_menu_add**: Add item
  - **@android:drawable/ic_menu_edit**: Edit item
  - **@android:drawable/ic_input_add**: Add new item
  - **@android:drawable/ic_delete**: Remove / clear
  - **@android:drawable/ic_menu_search**: Search action
  - **@android:drawable/ic_menu_save**: Save action
  - **@android:drawable/ic_menu_agenda**: Calendar / agenda
  - **@android:drawable/ic_menu_compass**: Navigation / location
  - **@android:drawable/ic_menu_share**: Share data
  - **@android:drawable/ic_menu_info_details**: Info dialog
  - **@android:drawable/ic_menu_close_clear_cancel**: Cancel / close
  - **@android:drawable/ic_menu_myplaces**: Home / location
  - **@android:drawable/ic_menu_camera**: Camera
  - **@android:drawable/ic_menu_gallery**: Gallery or media

- **@android:drawable/ic_menu_call**: Call / contact
- **@android:drawable/ic_menu_manage**: Manage list
- **@android:drawable/ic_dialog_alert**: Alert or warning dialog
- **@android:drawable/ic_dialog_info**: Info dialog icon

- **Dynamic resource names**: If your drawable image files are named like:

```
res/drawable/u1.png
res/drawable/u2.png
res/drawable/u3.png
...
res/drawable/u9.png
```

Then you can load them like this:

```
String imageName = "u" + i; // creates u1, u2, ..., u9
int resID = getResources().getIdentifier(imageName,
    ↪ "drawable", getPackageName());
```

- **Built-in layout resource**: Android provides a set of common, basic layouts so you don't need to create XML files for simple lists.
  - **android.R.simple_list_item_1**: It is basically a single TextView, with predefined padding and styling. The XML looks something like

```
<TextView xmlns:android="http://schemas.android.com/apk
    ↪ /res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:minHeight="?android:attr/listPreferredItemH
    ↪ eightSmall"
    android:gravity="center_vertical"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:textAppearance="?android:attr/textAppearanc
    ↪ eListItemSmall" />
```

    Perfect for simple `ArrayAdapter<String>` lists.
  - **android.R.layout.simple_spinner_item**: Used for the selected item in a Spinner. Also just one TextView

```
<TextView
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center_vertical"
    android:padding="8dp" />
```

Suitable for

```
new ArrayAdapter(this,
    ↪ android.R.layout.simple_spinner_item, data);
```

- **android.R.layout.simple_dropdown_item_1line**: Used for dropdown lists:
  - AutoCompleteTextView
  - Spinner dropdowns

  Again, a single TextView:

```
<TextView
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:singleLine="true"
    android:padding="8dp" />
```

  Used like this

```
adapter.setDropDownViewResource(android.R.layout.simple
    ↪ _dropdown_item_1line);
```

- **Other Common Built-In Layouts**:
  - **simple_list_item_1**: One TextView
  - **simple_list_item_2**: Two stacked TextViews (text1 and text2)
  - **simple_list_item_checked**: List item with checkmark
  - **simple_list_item_activated_1**: For activated/selected states
  - **simple_spinner_item**: Selected Spinner item
  - **simple_spinner_dropdown_item**: Dropdown row view
  - **simple_dropdown_item_1line**: Single-line dropdown suggestion
- **android.R.id.content**: special, system-defined ID that refers to the root view of your Activity's window—specifically, the top-level ViewGroup that holds everything inside your Activity's UI.

  This is the view that holds whatever layout you set using:

```
setContentView(...)
```

  So,

```
setContentView(R.layout.activity_main)
View layout = findViewById(android.R.id.content)
```

Puts the root layout for activity_main in the `layout` variable.

- **Unit argument in java**: When a method like TextViews `setTextSize(int unit, float size)` requires a unit parameter, we can used **TypedValue** constants like
  - **int COMPLEX_UNIT_DIP**: TYPE_DIMENSION complex unit: Value is Device Independent Pixels.
  - **int COMPLEX_UNIT_SP**: TYPE_DIMENSION complex unit: Value is a scaled pixel.
  - **int COMPLEX_UNIT_PX**: TYPE_DIMENSION complex unit: Value is raw pixels.

  For example,

```
0  tv.setTextSize(TypedValue.COMPLEX_UNIT_SP, fontSize);
```

2.3 Screen size, resolution, density, pixels, and dips (dp)

- **Screen size**: Screen size is measured diagonally from one corner of the display to the opposite corner. You measure from the bottom-left corner to the top-right corner (or bottom-right to top-left), the measurement is in inches.
- **Screen resolution**: Refers to the total number of pixels on the screen (width $\times$ height). Describes how many pixels the screen contains, not how big they are physically.
- **Screen density**: Refers to how tightly the pixels are packed into each inch of the screen, measured in dots per inch (dpi) or pixels per inch (ppi). Higher density = smaller pixels = sharper image. If a screen has a density of 300 dpi, then there are 300 pixels in one inch.

  The density of a screen depends on the size of the screen and the screens resolution.

$$\text{dpi} \approx \text{ppi} = \frac{\sqrt{\text{width}^2 + \text{hight}^2}}{\text{screen size}}.$$

  So, if we increase our screen resolution (keep the screen size constant), the screen density increases. If we increase our screen size (keep the resolution constant), our screen density decreases.

  Note: Screen density is measured in "dots per inch" (dpi) or pixels per inch (ppi). In most modern screens, a "dot" = a pixel, so dpi $\approx$ ppi.

- **Pixels**: The smallest dot of light on a screen, screens are made up of thousands or millions of pixels, more pixels equals more detail.

  You can calculate the size of a pixel with

$$\text{px size (in)} = \frac{1}{\text{ppi}}.$$

- **Density independent pixels (dp or dip)**: A virtual unit used in Android to keep UI elements the same physical size across devices. 1 dp = 1 pixel on a 160 dpi screen (the baseline screen density).

$$\text{px} = \text{dp}\left(\frac{\text{dpi}}{160}\right).$$

  Let $\rho =$ density factor $= \frac{\text{dpi}}{160}$, then

$$\begin{aligned} \text{px} &= \text{dp} \cdot \rho \\ \text{dp} &= \text{px}/\rho. \end{aligned}$$

  160 dpi (dots per inch) is considered the "baseline" or "medium density" screen in Android. This density is called mdpi. Historically, early Android phone screens (like the first HTC/Google phones) had around 160 dpi.

  Suppose you develop an app while working on a 360 dpi device, then the density factor is $\rho = \frac{360}{160} = 2$. Thus, for one pixel,

$$\text{dp} = \text{px}/\rho = 1/2.$$

So, 1 dp $= 2$ px . Now, suppose our device is now 720 dpi. Then, $\rho = 4$ and for one pixel,

$$\text{dp} = 1/4.$$

So, 1 dp $= 4$ px . Thus, as our density increases, the number of pixels per dp increases. As our density decreases, the number of pixels per dp decreases.

- **Getting screen resolution**:

```
Point p;
getWindowManager().getDefaultDisplay.getSize(size)
int width = p.x; int height = p.y;
```

- **Getting screen density in java**:

```
float rho = getResources().getDisplayMetrics.density; //
↪ dpi/160
```

- **Testing for screen sizes in java**: We can get screen size information through a Configuration object

```
Configuration config = getResources().getConfiguration();
```

- **Get height of the action bar**

```
import android.util.TypedValue;
...

TypedValue typedValue = new TypedValue();
if (getTheme().resolveAttribute(android.R.attr.actionBarSize
↪ , typedValue, true)) {
    int actionBarHeight =
    ↪ TypedValue.complexToDimensionPixelSize(
    typedValue.data, getResources().getDisplayMetrics());
}
```

- **Move root layout under the actionbar**: For example, if a GridLayout is the root layout

```
ViewGroup.MarginLayoutParams mlp = new
↪ ViewGroup.MarginLayoutParams(
    ViewGroup.LayoutParams.MATCH_PARENT,
    ViewGroup.LayoutParams.WRAP_CONTENT
);
mlp.setMargins(0,actionBarHeight*2,0,0);
```

2.4 Units

- **dp**: stands for density-independent pixels, or "dips" for short.

  The most common unit for layout dimensions (width, height, margins, padding). Scales with screen density so UI looks consistent across devices.

- **sp**: stands for scalable pixels. Maybe we can call them "sips"?

  Primarily for text size. Like dp, but also respects the user's font size settings (accessibility).

- **px (pixels)**: Actual screen pixels. Avoid using directly because it doesn't scale across different densities.

- **pt (points):** 1/72 of an inch. Rarely used, but supported.

- **in (inches)**: Physical size in inches (based on the screen's dpi).

- **mm (millimeter)**: Physical size in millimeters.

2.5 Files

- **AndroidManifest.xml**: The app's blueprint. Declares package name, permissions, minimum SDK, app components (activities, services, etc.), and the launcher activity.
- **MainActivity.java**: The main Java class that runs when the app starts. Controls app logic and connects the UI (XML layouts) with backend code.
- **activity_main.xml**: The layout file for MainActivity. Defines the UI elements (buttons, text, etc.) using XML.
- **colors.xml**: Central place for defining reusable color values. Used for themes, buttons, backgrounds, etc
- **strings.xml**: Stores all text strings (app name, labels, messages). Helps with reusability and localization (multi-language support).
- **dimens.xml**: Defines dimensions like margins, padding, font sizes (e.g., 16dp). Ensures consistent spacing and scaling.
- **themes.xml**: Holds styles and themes (colors, fonts, appearances) applied app-wide or to individual components.

2.6 The Manifest (AndroidManifest.xml)

- **<application>**: The <application> tag represents the entire Android app. It contains global settings for the app and includes all major components like activities, services, receivers, and providers.

  Specifies app-level settings like:

  - App icon (android:icon)
  - App name (android:label)
  - Default theme (android:theme)
  - Backup rules, RTL support, permissions, etc.

  Acts as a container for all components (<activity>, <service>, etc.)

- **<activity>**: An <activity> represents a single screen / page in your application — like an activity in Java corresponds to a UI screen. Each <activity> must be declared inside <application> so Android knows it exists.

- **android:configChanges**: In Android, the configChanges attribute in the AndroidManifest.xml is used to tell the system which configuration changes (like screen rotation, language change, keyboard type, etc.) your Activity will handle manually, instead of letting Android destroy and recreate it automatically.

  When certain configuration changes happen (like rotating the screen), Android by default:

  - Destroys the current Activity.
  - Recreates it with the new configuration (restarts onCreate(), etc.).
  - This allows the layout to be reloaded for the new screen size or orientation.

  By adding configChanges to your Activity in the manifest:

```
<activity
    android:name=".MainActivity"
    android:configChanges="orientation|screenSize">
</activity>
```

  You are telling Android "Don't recreate the Activity when orientation or screen size changes - I will handle it myself."

  So instead of restarting the Activity, it will call:

```java
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);

    if (newConfig.orientation ==
    ↪  Configuration.ORIENTATION_LANDSCAPE) {
        // Handle landscape changes
    } else if (newConfig.orientation ==
    ↪  Configuration.ORIENTATION_PORTRAIT) {
        // Handle portrait changes
    }
}
```

The configuration changes are

- **orientation**: Screen rotated (portrait ↔ landscape)
- **screenSize**: Screen size changes (usually with orientation)
- **keyboardHidden**: Hardware keyboard opened/closed
- **layoutDirection**: RTL (Arabic/Hebrew) vs LTR languages
- **uiMode:** Dark mode, car mode, TV mode
- **locale**: Language changes
- **fontScale**: System font size changes
- **density:** Screen pixel density changes
- **keyboard**: Keyboard type changes (physical/virtual)

2.7 themes.xml (items)

- **Color attributes**
  - **colorPrimary**: Main brand color; app bar, button, toggle background
  - **colorPrimaryVariant**: Darker/lighter version for emphasis (status bar, etc.)
  - **colorOnPrimary**: Text/icon color on top of primary surfaces
  - **colorSecondary**: Accent color used for FABs, toggles, etc.
  - **colorSecondaryVariant**: Variant for the secondary color
  - **colorOnSecondary**: Text/icon color drawn over secondary surfaces
  - **colorTertiary / colorOnTertiary**: Used in Material 3 themes for additional tonal roles
  - **colorSurface**: Background color of surfaces like cards, menus
  - **colorOnSurface**: Text/icon color on surfaces
  - **colorBackground**: Activity window background
  - **colorOnBackground**: Text/icon color on backgrounds
  - **colorError**: Used for error indicators, text fields, etc.
  - **colorOnError**: Text/icon color on error backgrounds
  - **colorOutline**: Divider and border color (Material 3)
  - **android:statusBarColor**: Status bar color (system UI)
  - **android:navigationBarColor**: Navigation bar color
  - **android:windowBackground**: Root background for the entire window
- **Typography and text appearance**
  - **android:fontFamily**: Default font for the app
  - **textAppearanceBody1**: Default text style for body text
  - **textAppearanceBody2**: Smaller body text style
  - **textAppearanceButton**: Button text style
  - **textAppearanceHeadline1** → **Headline6**: Large to small headline text
  - **textAppearanceSubtitle1,**: textAppearanceSubtitle2 Subtitle text
  - **textAppearanceCaption**: Caption/small label text
  - **textAppearanceOverline**: Overline (uppercase) small text
  - **android:textColorPrimary**: Primary text color on light/dark backgrounds
  - **android:textColorSecondary**: Secondary text color
- **Shape and corner styling (Material Components)**
  - **shapeAppearanceSmallComponent**: Shape for small widgets like buttons
  - **shapeAppearanceMediumComponent**: Medium-sized components (e.g., text fields)
  - **shapeAppearanceLargeComponent**: Large components like cards
  - **shapeAppearanceCornerSmall**: Controls corner radii of shapes
  - **shapeAppearanceCornerExtraLarge**: Very rounded shapes (chips, FABs)
  - **shapeAppearanceOverlay**: Used for overlaying corner radius configs
- **Window and system bar attributes**

  - **android:windowBackground**: Background drawable or color for the window
  - **android:windowFullscreen**: Whether activity draws behind status bar
  - **android:windowLightStatusBar**: Use dark icons on light status bar
  - **android:statusBarColor**: Status bar color
  - **android:navigationBarColor**: Navigation bar color
  - **android:windowTranslucentStatus**: Make status bar translucent
  - **android:windowTranslucentNavigation**: Make navigation bar translucent
  - **android:windowLayoutInDisplayCutoutMode**: Handle notches (cutouts)
  - **android:windowContentOverlay**: Overlay drawable between title bar and content

- **Widget and control styles**: Each of these points to another <style> resource defining widget looks.
  - **buttonStyle**: Default style for <Button>
  - **buttonStyleSmall**: Small button variant
  - **android:buttonStyleToggle**: Toggle button style
  - **switchStyle**: Default style for switches
  - **checkboxStyle**: Default style for checkboxes
  - **radioButtonStyle**: Default style for radio buttons
  - **editTextStyle**: Style for EditText
  - **textInputStyle**: Style for TextInputLayout (Material)
  - **spinnerStyle**: Default dropdown/spinner
  - **toolbarStyle**: Default Toolbar style
  - **materialButtonStyle**: Default Material Button style
  - **materialCardViewStyle**: Default CardView style
  - **floatingActionButtonStyle**: Default FAB style
  - **snackbarStyle**: Snackbar appearance
  - **bottomNavigationStyle**: Bottom nav bar
  - **navigationViewStyle**: Drawer navigation view
  - **appBarLayoutStyle**: App bar layout
  - **chipStyle**: Default Chip
  - **chipGroupStyle**: Default ChipGroup

- **Theme structure and inheritance**
  - **android:colorAccent**: (Pre-Material) old accent color (replaced by colorSecondary)
  - **android:colorControlHighlight**: Ripple color for controls
  - **android:colorControlActivated**: Color of activated controls
  - **android:colorControlNormal**: Default tint for controls
  - **android:colorButtonNormal**: Default background for buttons
  - **android:colorEdgeEffect**: Glow color when overscrolling
  - **android:alertDialogTheme**: Theme for dialogs
  - **android:popupMenuStyle**: Style for popup menus

  - **android:listViewStyle**: Default ListView
  - **android:progressBarStyle**: Progress bar
  - **android:seekBarStyle**: SeekBar
  - **android:ratingBarStyle**: RatingBar
  - **android:editTextColor**: EditText color (older)

- **Material 3 (Material You) dynamic color attributes**: If you're using a Material 3 theme (Theme.Material3.*):
  - **colorPrimaryContainer**: Container for primary UI elements
  - **colorOnPrimaryContainer**: Text/icons on primary container
  - **colorSecondaryContainer, colorOnSecondaryContainer**: Same but for secondary surfaces
  - **colorTertiaryContainer, colorOnTertiaryContainer**: Third tonal role
  - **colorSurfaceVariant**: Background for secondary surfaces
  - **colorInverseSurface, colorOnInverseSurface**: For inverted surfaces
  - **colorOutlineVariant**: Muted divider color
  - **colorScrim**: Used for modal overlays
  - **android:colorErrorContainer**: Error background tone
  - **android:colorOnErrorContainer**: Foreground on error container

2.8 XML tags

- **XML Declaration**: Android's resource compiler can usually parse XML without it. But it is strongly recommended to include it for consistency and to avoid encoding issues

```
<?xml version="1.0" encoding="utf-8"?>
```

- **Resources**: The root element of an XML resource file in the res/values/ directory (like strings.xml, colors.xml, styles.xml, etc.).

```
<resources>
    ...
</resources>
```

- **String**:

```
<string name=""> ... </string>
```

- **Color**:

```
<color name=""> hex </color>
```

- **Dimen**:

```
<dimen name=""> ... </dimen>
```

- **Manifest**

```
<manifest>
    ...
</manifest>
```

- **Style**: used to define styles and themes for your Android app

```
<style name="" parent="">
    <item name=""> ... </item>
</style>
```

2.9 Components

- **ConstraintLayout**: In Android Studio, ConstraintLayout is a powerful and flexible layout manager used to design UIs. It's often the default choice in modern Android projects.

  It is a **ViewGroup** (container) that positions and sizes its child views based on constraints you define.

  - **Constraints:** Each view needs at least one horizontal and one vertical constraint (e.g., align left to parent, center in parent, align top to another view).
  - **No deep nesting:** Unlike LinearLayout or RelativeLayout, you can achieve complex designs without nesting multiple layouts, which improves performance.
  - **Flexible positioning:** You can center, chain, bias (percent-based positioning), or even set aspect ratios.

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
...
tools:context=".MainActivity">
    ...
</androidx.constraintlayout.widget.ConstraintLayout>
```

  Note: xmlns stands for **XML Namespace**.

  `tools:context=".MainActivity"` tells the Layout Editor which Activity will load this layout. It's a design-time hint for Android Studio (not used at runtime).

- **TextView**: is a basic Android UI widget used to display text to the user. It's read-only by default (unlike EditText, which allows input). It can show plain text, styled text, or even links.

```
<TextView ...>
    ...
</TextView
```

- **EditText**: Subclass of TextView that allows the user to enter and edit text. It's the standard widget for text input in Android (like forms, search boxes, chat inputs)

```
<EditText ...>
...
</EditText>
```

2.10 Managing orientation

- **Manage orientation**: To manage an app's orientation on a device, we must
  1. retrieve the screen dimensions of a device dynamically,
  2. retrieve the height of both the status and action bars (if there),
  3. be able to detect orientation changes on the device, and
  4. manage the UI of the app when the user rotates the device.
- **Device rotations**: When we run an app and rotate the device that the app is running on, the UI may or may not rotate.

  Some apps, in particular games, are best run in one orientation only, often horizontal. Sometimes, we want to build an app that works in both horizontal and vertical orientations.
- **Locking apps to only one orientation**:
  - **Portrait only**:

```
<!-- AndroidManifest.xml -->
<activity
    android:name=".MainActivity"
    android:screenOrientation="portrait" />
```

    To allow reverse landscape (upside down),

```
<activity
    android:name=".MainActivity"
    android:screenOrientation="sensorPortrait" />
```

    Or in code (at runtime):

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION
↪ _PORTRAIT);
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION
↪ _SENSOR_PORTRAIT);
```

  - **Reverse portrait only**:

```
<activity
   android:name=".MainActivity"
   android:screenOrientation="reversePortrait" />
```

    In java,

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION
↪ _REVERSE_PORTRAIT);
```

– **Landscape only**

```
<activity
    android:name=".MainActivity"
    android:screenOrientation="landscape" />
```

If you want to allow both landscape directions (normal + reverse), use:

```
<activity
    android:name=".MainActivity"
    android:screenOrientation="sensorLandscape" />
```

In java,

```
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION
↪ _LANDSCAPE);
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION
↪ _SENSOR_LANDSCAPE);
```

– **Reverse landscape**

```
<activity
   android:name=".MainActivity"
   android:screenOrientation="reverseLandscape" />
```

In java,

```
ActivityInfo.SCREEN_ORIENTATION_REVERSE_LANDSCAPE
```

- **Allow any orientation**: To allow only landscape and portrait,

```
<activity
    android:name=".MainActivity"
    android:screenOrientation="sensor" />
```

This allows both portrait and landscape. If we want to include reverse landscape and reverse portrait,

```
<activity
   android:name=".MainActivity"
   android:screenOrientation="fullSensor" />
```

- **Default behavior**: By default, neither sensor nor fullSensor is explicitly used. If you do not set android:screenOrientation in the manifest and do not call setRequestedOrientation() in code, then Android uses:

```
android:screenOrientation="unspecified"
```

  The app follows the device orientation sensor, but only within the orientations allowed by the user's system settings, typically supports portrait and landscape (not reverse portrait/landscape unless the device/user allows it). So, it behaves similar to sensor — not fullSensor.

- **Getting resolution**: We can fill a `Point` object using the `getSize(Point p)` method in the Display object.

```
Point size = new point();
getWindowManager().getDefaultDisplay.getSize(size)
int width = size.x;
int height = size.y;
```

- **Getting resolution in dp**: Through a Configuration object,

```
Configuration config = getResources().getConfiguration();
int width_dp = config.screenWidthDp;
int height_dp = config.screenHeighthDp;
```

- **Getting density**: We can get the screen density factor ρ (logical pixel density) through the DisplayMetrics object

```
DisplayMetrics metrics = getResources().getDisplayMetrics();
float rho = metrics.density;
```

- **Testing for size type**: First, we get a Configuration object

```
Configuration config = getResources().getConfiguration();
```

  Then,

```
if (config.isLayoutSizeAtLeast(Configuration.SCREENLAYOUT_SIZE_XLARGE))
    Log.w(MA, "Extra large size screen");
else if (config.isLayoutSizeAtLeast(Configuration.SCREENLAYOUT_SIZE_LARGE))
    Log.w(MA, "Large size screen");
else if (config.isLayoutSizeAtLeast(Configuration.SCREENLAYOUT_SIZE_NORMAL))
    Log.w(MA, "Normal size screen");
else if (config.isLayoutSizeAtLeast(Configuration.SCREENLAYOUT_SIZE_SMALL))
    Log.w(MA, "Small size screen");
else
    Log.w(MA, "Unknown size screen");
```

- **Getting orientation information**: We get a Configuration object reference through the Resources object reference, the Configuration object holds information about orientation

```
Configuration config = getResources().getConfiguration();
int orientation = config.orientation;
```

- **Managing orientation change outside onCreate()**: By default, method onCreate() of the MainActivity.java class is called when the user rotates the device. However, method onCreate() typically does more than just handling orientation changes. Thus, it could be a waste of CPU resources to let method onCreate() execute every time the user rotates the device.

  We can specify `android:configChanges="orientation|screenSize"` to handle rotations manually, instead of the default behavior.

  Then, we can override `onConfigurationChanged(Configuration newConfig)`. For example,

```
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    Log.w(MA, "Height: " + newConfig.screenHeightDp);
    Log.w(MA, "Width: " + newConfig.screenWidthDp);

    Log.w(MA, "Orientation: " + newConfig.orientation);
    if (newConfig.orientation ==
     ↪  Configuration.ORIENTATION_LANDSCAPE)
        Log.w(MA, "Horizontal position");
    else if (newConfig.orientation ==
     ↪  Configuration.ORIENTATION_PORTRAIT)
        Log.w(MA, "Vertical position");
    else
        Log.w(MA, "Undetermined position");
}
```

Method onCreate() is only called when the user starts the app and no longer called when the user rotates the device.

- **Display correct UI based on orientation**: We can do it one of three ways.
  1. We can create one layout XML file per orientation and inflate it whenever the user rotates the device
  2. We can create one layout XML file for both orientations and modify the characteristics of some of the UI components whenever the user rotates the device
  3. We can manage layouts 100% by code and make the appropriate modifications whenever the user rotates the device
- **One layout file per orientation**: We can have `activity_main.xml` for portrait, and `activity_main_landscape.xml` for landscape. In the manifest, we write

```
android:configChanges="orientation|screenSize"
```

Then in java, we can override the activity method `onConfigurationChanged`

```
public void onConfigurationChanged(Configuration newConfig) {
    Log.w("MainActivity", "Inside onConfigurationChanged");
    super.onConfigurationChanged(newConfig);
    modifyLayout(newConfig);
}
```

This function will be called when the configuration changes. Our modifyLayout function could look like

```
public void modifyLayout(Configuration newConfig) {
    if (newConfig.orientation ==
    ↪  Configuration.ORIENTATION_LANDSCAPE)
        setContentView(R.layout.activity_main_landscape);
    else if (newConfig.orientation ==
    ↪  Configuration.ORIENTATION_PORTRAIT)
        setContentView(R.layout.activity_main);
}
```

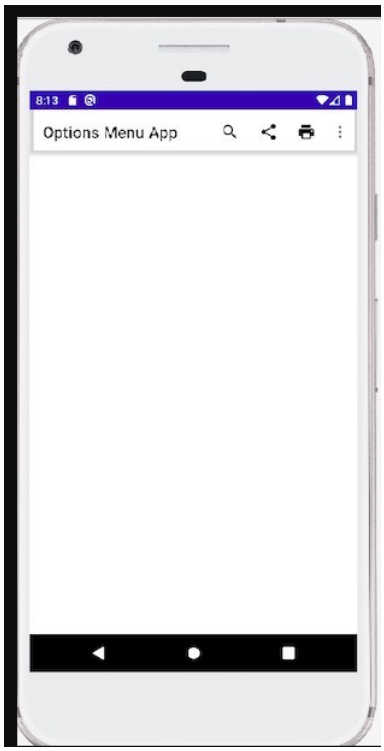
We also need our `onCreate` to have a few details as well,

```
protected void onCreate(Bundle savedInstanceState)
{
    Log.w("MainActivity", "Inside onCreate");
    super.onCreate(savedInstanceState);
    Configuration config = getResources().getConfiguration();
    modifyLayout(config);
}
```

This way we inflate the correct layout when the app first starts.

2.11 Menus

- **Menu Resource directory**: Menus are defined in `res/menu`
- **Menu types**
  - **Options menu**: Defined in the action bar / app bar, or in the overflow menu if there's no room.
  - **Context menu**: Appears when the user long-presses an item. Appears next to the pressed item OR as a floating context bar
  - **Popup menu**: A tiny menu that pops up anchored to a button, Triggered by tapping a specific UI element (not long-press)
- **Options menu**



We need to override the `onCreateOptionsMenu` method

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}
```

The menu parameter is an empty Menu object created by Android. It represents the Action Bar options menu that will be shown to the user.

Your job in this method is to add items into this menu, usually done by inflating XML into it

Returning true means yes, we successfully created and want to show this menu.

To handle clicks, we override

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.action_settings) {
        // action here
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Calling the super version lets the default Android behavior run when your code doesn't handle the menu click. If it's not your menu item, let the Android system decide what to do.

- **Context Menu**: We first need to register the view

```
registerForContextMenu(myView);
```

Then, override

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
ContextMenu.ContextMenuInfo menuInfo) {
    getMenuInflater().inflate(R.menu.context_menu, menu);
}
```

To handle clicks, override

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.action_delete) {
        // do delete
        return true;
    }
    return super.onContextItemSelected(item);
}
```

- **Popup menu**: First, create in code where needed

```
PopupMenu popup = new PopupMenu(this, myButton);
popup.getMenuInflater().inflate(R.menu.popup_menu,
    ↪  popup.getMenu());
popup.show();
```

Handle clicks using a listener

```
popup.setOnMenuItemClickListener(item -> {
    if (item.getItemId() == R.id.action_share) {
        // do share
        return true;
    }
    return false;
});
```

- **Menu attributes**
  - **android:id=”Resource ID”**:. A unique resource ID. To create a new resource ID for this item, use the form: ”@+id/name”. The plus symbol indicates that this is created as a new ID.
  - **android:title=”String resource”**:. The menu title as a string resource or raw string.
  - **android:titleCondensed=”String resource”**:. A condensed title as a string resource or a raw string. This title is used for situations in which the normal title is too long.
  - **android:icon=”Drawable resource”**:. An image to be used as the menu item icon.
  - **android:onClick=”Method name”**:. The method to call when this menu item is clicked. The method must be declared in the activity as public. It accepts a MenuItem as its only parameter, which indicates the item clicked. This method takes precedence over the standard callback to onOptionsItemSelected(). See the example at the end of this page.
  - **android:showAsAction=”Keyword.”**: When and how this item appears as an action item in the app bar. A menu item can appear as an action item only when the activity includes an app bar. Valid values:
    - **ifRoom**: Only place this item in the app bar if there is room for it. If there isn’t room for all the items marked ”ifRoom”, the items with the lowest orderInCategory values are displayed as actions, and the remaining items are displayed in the overflow menu.
    - **withText**: Also include the title text (defined by android:title) with the action item. You can include this value along with one of the others as a flag set by separating them with a pipe |.
    - **never**: Never place this item in the app bar. Instead, list the item in the app bar’s overflow menu.
    - **always**: Always place this item in the app bar. Avoid using this unless it’s critical that the item always appear in the action bar. Setting multiple items to always appear as action items can result in them overlapping with other UI in the app bar.

    - **collapseActionView**: The action view associated with this action item (as declared by android:actionLayout or android:actionViewClass) is collapsible. Introduced in API level 14.

- **android:actionLayout="Layout resource"**:. A layout to use as the action view.
- **android:actionViewClass="Class name"**:. A fully-qualified class name for the View to use as the action view. For example, "android.widget.SearchView" to use SearchView as an action view.
- **android:actionProviderClass="Class name"**:. A fully qualified class name for the ActionProvider to use in place of the action item. For example, "android.widget.ShareActionProvider" to use ShareActionProvider.
- **android:alphabeticShortcut="Char."**: A character for the alphabetic shortcut key.
- **android:numericShortcut="Integer."**: A number for the numeric shortcut key.
- **android:alphabeticModifiers="Keyword."**: A modifier for the menu item's alphabetic shortcut. The default value corresponds to the Control key. Valid values:
    - **META**: Corresponds to the Meta meta key.
    - **CTRL**: Corresponds to the Control meta key.
    - **ALT**: Corresponds to the Alt meta key.
    - **SHIFT**: Corresponds to the Shift meta key.
    - **SYM**: Corresponds to the Sym meta key.
    - **FUNCTION**: Corresponds to the Function meta key.

  You can use the setAlphabeticShortcut() method to set the attribute values programmatically. For more information about the alphabeticModifier attribute
- **android:numericModifiers="Keyword"**:. A modifier for the menu item's numeric shortcut. The default value corresponds to the Control key. Valid values:
    - **META**: Corresponds to the Meta meta key.
    - **CTRL**: Corresponds to the Control meta key.
    - **ALT**: Corresponds to the Alt meta key.
    - **SHIFT**: Corresponds to the Shift meta key.
    - **SYM**: Corresponds to the Sym meta key.
    - **FUNCTION**: Corresponds to the Function meta key.

  You can use the setNumericShortcut() method to set the attribute values programmatically. For more information about the numericModifier attribute
- **android:checkable="Boolean"**:. True if the item is checkable.
- **android:checked="Boolean"**:. True if the item is checked by default.
- **android:visible="Boolean"**:. True if the item is visible by default.
- **android:enabled="Boolean"**:. True if the item is enabled by default.
- **android:menuCategory="Keyword"**:. Value corresponding to the Menu CATEGORY_* constants, which define the item's priority. Valid values:
    - **container**: For items that are part of a container.
    - **system**: For items that are provided by the system.
    - **secondary**: For items that are user-supplied secondary (infrequently used) options.

    - **alternative**: For items that are alternative actions on the data that is currently displayed.
  - **android:orderInCategory="Integer"**:. The order of importance of the item within a group.

- **<group>**: A menu group, to create a collection of items that share traits, such as whether they are visible, enabled, or selectable. Contains one or more <item> elements. Must be a child of a <menu> element.

- **<group> attributes**:
  - **android:id="Resource ID"**: A unique resource ID. To create a new resource ID for this item, use the form: "@+id/name". The plus symbol indicates that this is created as a new ID.
  - **android:checkableBehavior="Keyword"**:. The type of selectable behavior for the group. Valid values:
    - **none**: Not selectable.
    - **all**: All items can be selected (use checkboxes).
    - **single**: Only one item can be selected (use radio buttons).
  - **android:visible="Boolean"**:. True if the group is visible.
  - **android:enabled="Boolean"**:. True if the group is enabled.
  - **android:menuCategory="Keyword"**:. Value corresponding to the Menu CATEGORY_* constants, which define the group's priority. Valid values:
    - **container**: For groups that are part of a container.
    - **system**: For groups that are provided by the system.
    - **secondary**: For groups that are user-supplied secondary (infrequently used) options.
    - **alternative**: For groups that are alternative actions on the data that is currently displayed.
  - **android:orderInCategory**: Integer. The default order of the items within the category.

2.12 SQLite

- **Extending SQLiteOpenHelper**: We can create a `DatebaseManager` class file, in which we extend `SQLiteOpenHelper`. That's Android's built-in helper for:
  - **Creating tables**: onCreate(SQLiteDatabase db)
  - **Upgrading schema versions**: onUpgrade(SQLiteDatabase db, int oldV, int newV)
  - **Opening / closing the DB safely**: handled by the parent class

  Android automatically calls onCreate() the first time your app accesses the database.

- **Define name and version**: For example,

```
private static final String DATABASE_NAME = "recordDB";
private static final int DATABASE_VERSION = 1;
```

  Stored in your app's private folder:

```
/data/data/<your-package>/databases/recordDB
```

  Version controls schema changes; bumping it triggers onUpgrade().

- **Table creation**: For example,

```
public void onCreate(SQLiteDatabase db) {
    String sqlCreate = "create table record(" +
        "id integer primary key autoincrement, " +
        "name text, price real)";
    db.execSQL(sqlCreate);
}
```

  Called automatically the first time the database is created. Builds one table named record with:
  - **id**: primary key (auto-increment)
  - **name**: text column
  - **price**: real (floating-point number)

- **Upgrading the schema**: For example,

```
public void onUpgrade(SQLiteDatabase db, int oldVersion, int
    ↪ newVersion) {
    db.execSQL("drop table if exists record");
    onCreate(db);
}
```

When you increment DATABASE_VERSION, Android calls this. It drops the old table and recreates it. (In production you'd use ALTER TABLE instead of dropping data.)

- **Inserting data**: For example,

```
public void insert(Record record) {
    SQLiteDatabase db = this.getWritableDatabase();
    String sqlInsert = "insert into record values(null, '" +
                       record.getName() + "', '" +
                       ↪  record.getPrice() + "')";
    db.execSQL(sqlInsert);
    db.close();
}
```

Opens a writable connection, Executes a raw SQL INSERT statement, Uses null for the auto-incrementing ID, Closes the database

Note: In real apps, you'd use ContentValues to avoid SQL injection.

- **Deleting data**: For example,

```
public void deleteById(int id) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.execSQL("delete from record where id = " + id);
    db.close();
}
```

Removes one row where the ID matches.

- **Updating data**

```
public void updateById(int id, String name, double price) {
    SQLiteDatabase db = this.getWritableDatabase();
    db.execSQL("update record set name = '" + name + "',
    ↪  price = '" + price + "' where id = " + id);
    db.close();
}
```

Changes both name and price columns for the selected record.

- **Querying one record**

```
public Record selectById(int id) {
    String sqlQuery = "select * from record where id = " +
    ↪  id;
    SQLiteDatabase db = this.getWritableDatabase();
    Cursor cursor = db.rawQuery(sqlQuery, null);

    Record record = null;
    if (cursor.moveToFirst())
        record = new Record(
            Integer.parseInt(cursor.getString(0)),
            cursor.getString(1),
            cursor.getDouble(2));
    return record;
}
```

- Executes a SELECT query
- Returns a Cursor (a pointer over result rows)
- Reads columns using indices (0=id, 1=name, 2=price)
- Builds and returns a Record object

A Cursor in Android is a data access object that points to the result set (rows and columns) returned from a database query. It works like an iterator that lets you move across each row of the query result and read each column's value.

- sqlQuery = a plain SQL command to get 1 row (SELECT * FROM record WHERE id = ?).
- db.rawQuery(sqlQuery, null) executes the SQL and returns a Cursor.
- The cursor now contains the query result (in memory, inside SQLite).

Important methods

- **moveToFirst()** Moves the cursor to the first row of the result set. Returns false if there are no rows.
- **moveToNext()** Moves the cursor forward one row. Returns false if there are no more rows.
- **moveToPrevious()** Moves the cursor back one row.
- **moveToLast()** Moves the cursor to the last row.
- **move(int offset)** Moves the cursor relative to its current position.
- **moveToPosition(int position)** Moves to an exact row index (0-based). Returns false if out of range.
- **isFirst() / isLast()** Returns true if the cursor is at the first/last row.
- **isBeforeFirst() / isAfterLast()** Returns true if the cursor hasn't started or has finished iterating.
- **getPosition()** Returns the current row index.
- **getInt(int columnIndex)**: Reads an int value.
- **getLong(int columnIndex)**: Reads a long.
- **getDouble(int columnIndex)**: Reads a double.
- **getFloat(int columnIndex)**: Reads a float.
- **getString(int columnIndex)**: Reads a String.

- **getBlob(int columnIndex)**: Reads a byte[] (for images or binary data).
- **isNull(int columnIndex)**: Returns true if the column's value is NULL.
- **getColumnCount()**: Returns number of columns in the result.
- **getColumnName(int index)**: Returns the name of the column at a given index.
- **getColumnNames()**: Returns a String[] of all column names.
- **getColumnIndex(String columnName)**: Returns the column's index (or -1 if not found).
- **getColumnIndexOrThrow(String columnName)**: Same as above, but throws IllegalArgumentException if not found.
- **getCount()**: Returns how many rows are in the result set.
- **isClosed()**: Returns true if the cursor is already closed.
- **getPosition()**: Returns current row index (0 = first).
- **close()**: Frees database and native resources. Always call when done!

- **Query all records**

```
0  public ArrayList<Record> selectAll() {
1      String sqlQuery = "select * from record";
2      SQLiteDatabase db = this.getWritableDatabase();
3      Cursor cursor = db.rawQuery(sqlQuery, null);
4
5      ArrayList<Record> records = new ArrayList<>();
6      while (cursor.moveToNext()) {
7          Record current = new Record(
8              Integer.parseInt(cursor.getString(0)),
9              cursor.getString(1),
10             cursor.getDouble(2));
11         records.add(current);
12     }
13     db.close();
14     return records;
15 }
```

- **Impotant SQLiteDatabase methods**
  - **getReadableDatabase()**: Opens the database in read-only mode.
  - **getWritableDatabase()**: Opens the database in read/write mode (creates it if needed).
  - **close()**: Closes the database connection.
  - **isOpen()**: Returns true if the DB is currently open.
  - **getPath()**: Returns the absolute file path of the database.
  - **execSQL(String sql)**: Executes a single SQL statement (no results returned). For example: db.execSQL("DELETE FROM record WHERE id=5");
  - **insert(String table, String nullColumnHack, ContentValues values)**: Inserts a new row. Returns the new row ID.
  - **update(String table, ContentValues values, String whereClause, String[] whereArgs)**: Updates existing rows matching a condition. Returns number of rows affected.

- **delete(String table, String whereClause, String[] whereArgs)**: Deletes rows matching the condition. Returns number of rows deleted.
- **beginTransaction()**: Starts a new transaction.
- **setTransactionSuccessful()**: Marks the transaction to commit.
- **endTransaction()**: Ends the transaction (commits if marked successful, rolls back otherwise).
- **inTransaction()**: Returns true if currently inside a transaction.
- **execSQL(String sql)**: Run any SQL (CREATE, DROP, etc.).
- **getVersion()**: / setVersion(int version) Read or change DB version number.
- **getMaximumSize()**: / setMaximumSize(long numBytes) Check or set max DB file size.
- **needUpgrade(int newVersion)**: Returns true if the database version is lower than the given version.
- **isDatabaseIntegrityOk()**: Verifies database file integrity.
- **isReadOnly()**: Returns true if database is opened read-only.
- **isWriteAheadLoggingEnabled()**: Returns true if WAL mode is active.

2.13 Styles

- **How to style**: We can use many attributes to specify how a UI component will look. For example, we can specify the background color of a component.

  We can specify its text size and color and whether the text is centered or not. We can also specify the component's size and the padding inside it

  We use styles and themes to organize our project better

  Themes and styles enable us to separate the look and feel of a View from its content. This is similar to the concept of CSS (Cascading Style Sheets) in web design.

  We can define more styles in the file named themes.xml in the res > values directory.

- **Define a style** The syntax for defining a style is

```
<style name="nameOfStyle"
    [parent="styleThisStyleInheritsFrom"]>
    <item name="attributeName" parent="styleThisStyleInherit⌋
    ↪  sFrom">attributeValue</item>
    ...
</style>
```

  The (optional) parent attribute allows us to create a hierarchy of styles, i.e., styles can inherit from other styles.

- **Example style**: For example, this style specifies width, height, text size, and padding

```
<style name="TextStyle"
    parent="@android:style/TextAppearance">
    <item name="android:layout_width">wrap_content</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:textSize">32sp</item>
    <item name="android:padding">10dp</item>
</style>
```

- **Apply a style**: We give the style attribute to a component. For example,

```
<button style="@style/styleName" ...> ... </button>
```

- **TextView Styles**:
  - **android:layout_width**
  - **android:layout_height**
  - **android:layout_margin**
  - **android:layout_gravity**
  - **android:ellipsize**: how text is cut off (none, start, middle, end, marquee)

  - **android:singleLine**: (deprecated, use maxLines="1")
  - **android:textSize**: font size (e.g. 16sp)
  - **android:textColor**: text color (e.g. @color/black)
  - **android:textColorHint**: color of the hint text
  - **android:textColorHighlight**: color of text selection highlight
  - **android:textColorLink**: color of hyperlinks
  - **android:textStyle**: normal, bold, italic
  - **android:fontFamily**: font family (e.g. sans-serif, serif, or custom font from res/font/)
  - **android:typeface**: older way of setting (normal, sans, serif, monospace)
  - **android:lineSpacingExtra**: add extra space between lines
  - **android:lineSpacingMultiplier**: scale spacing between lines
  - **android:letterSpacing**: adjust space between characters
  - **android:gravity**: how text is positioned inside its box (top, bottom, left, right, center)
  - **android:textAlignment**: alignment relative to parent (gravity, center, viewStart, etc.)
  - **android:includeFontPadding**: extra font padding (default true)
  - **android:scrollHorizontally**: allow horizontal scroll if needed
  - **android:ems**: width in "M" units
  - **android:shadowColor**: color of text shadow
  - **android:shadowDx, android:shadowDy**: shadow offset
  - **android:shadowRadius**: shadow blur radius

- **EditText**: Since EditText is a subclass of TextView, it inherits all of TextView's styling attributes
  - **android:textCursorDrawable**: lets you set a custom drawable for the blinking cursor.
  - **android:textSelectHandle**: base selection handle drawable.
  - **android:textSelectHandleLeft**: left handle for text selection.
  - **android:textSelectHandleRight**: right handle for text selection.
  - **android:colorControlActivated**: (theme attr, but affects EditText focus underline in Material/AppCompat).
  - **android:colorControlNormal**: normal underline/tint when unfocused.
  - **android:colorControlHighlight**: ripple/highlight color when focused.

- **Button**: Button is another subclass of TextView, so it inherits all of TextView's styling attributes
  - **android:text**: label text.
  - **android:textSize**: text size (14sp, 18sp).
  - **android:textColor**: text color.
  - **android:textStyle**: normal, bold, italic.
  - **android:fontFamily**: custom font (@font/roboto_bold).
  - **android:letterSpacing**: adjust spacing between characters.

- **android:lineSpacingExtra / android:lineSpacingMultiplier**: line spacing.
- **android:textAllCaps**: force all caps (default true on Material buttons).
- **android:ellipsize**: how text is cut off if too long.
- **android:background**: drawable for button background (e.g. custom shape).
- **android:foreground**: optional overlay (e.g. ripple).
- **android:insetLeft, android:insetRight, android:insetTop, android:insetBottom**: padding inside button background (mostly for legacy Button).
- **android:padding, android:paddingStart, android:paddingEnd**: space inside button.
- **android:layout_width, android:layout_height**: sizing.
- **android:minWidth, android:minHeight**: minimum size (Material buttons have built-in minimums).
- **android:backgroundTint**: tint for the button background.
- **android:backgroundTintMode**: blend mode for tint.
- **android:drawableTint**: tint icons/drawables inside button.
- **android:drawableTintMode**: blending for icon tint.
- **android:drawableStart / android:drawableLeft**: icon at start.
- **android:drawableEnd / android:drawableRight**: icon at end.
- **android:drawableTop, android:drawableBottom**: icons above/below text.
- **android:drawablePadding**: space between icon and text
- **android:elevation**: z-depth shadow (Material design).
- **android:translationZ**: raised elevation when pressed.
- **android:shadowColor, android:shadowDx, android:shadowDy, android:shadowRadius**: text shadow.
- **app:cornerRadius**: rounded corners.
- **app:strokeColor**: outline color.
- **app:strokeWidth**: outline width.
- **app:icon**: set an icon.
- **app:iconPadding**: space between icon and text.
- **app:iconGravity**: where the icon appears (start, end, textStart, textEnd).
- **app:iconTint**: tint icon color.

2.14 Events

- **View (XML)**
  - **android:onClick**: name of a method in your Activity that gets called when the button is clicked

2.15 Java event listeners

- **Views:**
  - **void setOnClickListener(OnClickListener l)**: Registers a listener to be invoked when the view is clicked (short tap).
  - **void setOnLongClickListener(OnLongClickListener l)**: Registers a listener to be invoked when the view is long-pressed.
  - **void setOnFocusChangeListener(OnFocusChangeListener l)**: Sets a listener that is called whenever the view gains or loses input focus.
  - **void setOnTouchListener(OnTouchListener l)**: Sets a listener to receive touch events (e.g., finger down, move, lift) before they are processed by `onTouchEvent()`.
  - **void setOnKeyListener(OnKeyListener l)**: Sets a listener to receive key events (e.g., hardware button presses) before they are passed to `onKeyDown()` or `onKeyUp()`.
  - **boolean onKeyDown(int keyCode, KeyEvent event)**: Called when a hardware key is pressed down while the view has focus.
  - **boolean onKeyUp(int keyCode, KeyEvent event)**: Called when a hardware key is released while the view has focus.
  - **boolean onTouchEvent(MotionEvent event)**: Handles touch screen interaction with the view (default implementation supports clicks, scrolls, etc.).
  - **boolean onGenericMotionEvent(MotionEvent event)**: Handles non-touch input events such as mouse, joystick, or stylus actions.
  - **boolean onKeyPreIme(int keyCode, KeyEvent event)**: Called when a key event occurs before it reaches the input method (useful for intercepting Back button presses while an IME is visible).
  - **boolean onTrackballEvent(MotionEvent event)**: Handles trackball events (legacy input for older devices).
- **TextView**
  - **void addTextChangedListener(TextWatcher watcher)**: Registers a TextWatcher to receive callbacks when the text changes (before, during, or after editing).
  - **void removeTextChangedListener(TextWatcher watcher)**: Unregisters a previously added TextWatcher so it no longer receives callbacks.
  - **void setOnEditorActionListener(TextView.OnEditorActionListener l)**: Sets a listener to handle editor actions from the soft keyboard (e.g., pressing "Done", "Next", or "Search").
  - **void setOnClickListener(View.OnClickListener l)**: Assigns a listener to handle click events when the view is tapped.
  - **void setOnLongClickListener(View.OnLongClickListener l)**: Assigns a listener to handle long-click (press-and-hold) events on the view.
  - **void setOnFocusChangeListener(View.OnFocusChangeListener l)**: Sets a listener that is triggered when the view gains or loses input focus.
- **EditText**:
  - **void addTextChangedListener(TextWatcher watcher)**: Registers a listener for text changes.
  - **void removeTextChangedListener(TextWatcher watcher)**: Unregisters a text change listener.

  - **void setOnEditorActionListener(TextView.OnEditorActionListener l)**: Sets a listener for handling IME action events (e.g., pressing Enter, Done, Search).

- **Button:** Subclass of TextView, which is a subclass of View, so all the ones listed above

2.16 Java event listeners (2)

- **TextWatcher**: The `TextWatcher` interface, from the android.text package, provides three methods to handle key events.

  You can attach a TextWatcher to any view that implements Editable text content — meaning subclasses of TextView that allow editing.

```
0  import android.text.TextWatcher;
1  import android.text.Editable;
2  import java.lang.CharSequence;
3
4  .
5  .
6  .
7
8
9  private class TextChangeHandler implements TextWatcher {
10     public void beforeTextChanged(CharSequence s, int start,
       ↪  int count, int after )
11
12     public void onTextChanged(CharSequence s, int start, int
       ↪  before, int after)
13
14     public void afterTextChanged(Editable e)
15 }
```

  Where (`beforeTextChanged`)

  - **CharSequence s**: The text before the change.
  - **int start**: The position (index) in the text where the change will begin.
  - **int count**: How many characters are about to be replaced (i.e., how many will be removed).
  - **int after**: How many characters will replace the old ones (i.e., how many will be added).

  (onTextChanged)

  - **CharSequence s**: The text after the change (current state).
  - **int start**: The position in the text where the change happened.
  - **int before**: Number of characters that were replaced (removed).
  - **int count**: Number of new characters added.

  (afterTextChanged)

  - **Editable:** Represents the text content of the EditText after a change has occurred.

  Instantiate an instance of the class and attach it to a view with `.addTextChangedListener()`

- **onClick listener**: Create a private inner class that implements View.OnClickListener, and overrides the function onClick with signature

```
public void onClick(View v)
```

Create an instance of the inner class and use the function `setOnClickListener()` to add it to a view.

Or, use an anonymous inner class

```
myView.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        ...
    }
});
```

2.17 Code examples

- **Minimum code for controller**:

```
package com.example.test3;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

- **Base theme:**

```
<style name="Base.Theme.TipCalculator"
    parent="Theme.Material3.DayNight.NoActionBar">
    <!-- Customize your light theme here. -->
    <!-- <item
        name="colorPrimary">@color/my_light_primary</item>
        -->
</style>
```

- **GestureDetector.OnGestureListener**

```
public class MainActivity extends AppCompatActivity implements GestureDetector.OnGestureListener {

private GestureDetector gestureDetector;
private static final String TAG = "GESTURE";

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Simple full-screen view
    TextView tv = new TextView(this);
    tv.setText("Touch and scroll or fling");
    tv.setTextSize(30);
    setContentView(tv);

    // Create the GestureDetector using THIS as the listener
    gestureDetector = new GestureDetector(this, this);
}

// Forward all MotionEvents to the GestureDetector
@Override
public boolean onTouchEvent(MotionEvent event) {
    return gestureDetector.onTouchEvent(event);
}

// ----------------------------
// GestureDetector.OnGestureListener Methods
// ----------------------------

@Override
public boolean onDown(MotionEvent e) {
    Log.w(TAG, "onDown");
    return true;  // must return true to continue receiving events
}

@Override
public void onShowPress(MotionEvent e) {
    Log.w(TAG, "onShowPress");
}

@Override
public boolean onSingleTapUp(MotionEvent e) {
    Log.w(TAG, "onSingleTapUp");
    return true;
}

@Override
public boolean onScroll(MotionEvent e1, MotionEvent e2,
                        float distanceX, float distanceY) {
    Log.w(TAG, "onScroll: distanceX=" + distanceX
            + ", distanceY=" + distanceY);
    return true;
}

@Override
public void onLongPress(MotionEvent e) {
    Log.w(TAG, "onLongPress");
```

Java reference

3.1 Activity

- **Lifecycle methods**
  - **onCreate(Bundle)**: Initialize activity (UI layout, variables, listeners). Most important one.
  - **onStart()**: Activity is becoming visible.
  - **onResume()**: Activity is now in the foreground and interactive.
  - **onPause()**: Partially visible — pause resources, stop animations. called when Android starts or resumes another
  - **onStop()**: No longer visible — release resources.
  - **onRestart()**: Called before restarting after being stopped.
  - **onDestroy()**: Final cleanup before activity is removed from memory.
- **Transitions**
  - **overridePendingTransition(int enterAnim, int exitAnim)**: Overrides the default transition right after calling startActivity() or finish()
  - **finishAfterTransition()**: Finishes the Activity after shared-element transitions complete
  - **postponeEnterTransition()**: Waits before running transition (useful while loading images)
  - **startPostponedEnterTransition()**: Resumes the delayed enter transition
  - **setEnterSharedElementCallback(SharedElementCallback)**: Customize shared element mapping during enter transition
  - **setExitSharedElementCallback(SharedElementCallback)**: Same as above, but for exit transitions
- **Transition constants**
  - **TRANSITION_NONE**: No animation
  - **TRANSITION_OPEN**: When an Activity is opened
  - **TRANSITION_CLOSE**: When an Activity is closed
- **UI and layout management**
  - **setContentView(int layoutResId)**: Sets the XML layout for the Activity.
  - **findViewById(int id)**: Get references to UI elements.
- **Navigation and intents**
  - **startActivity(Intent)**: Launch another Activity.
  - **startActivityForResult(Intent, int)**: Launch an Activity and get returned data. (Deprecated in favor of ActivityResult API, but still widely used.)
  - **finish()**: Close the current Activity.
  - **getIntent()**: Retrieve the Intent that started the Activity.
- **State Preservation**: When configuration changes happen (like screen rotation):
  - **onSaveInstanceState(Bundle)**: Save UI state before destruction.

  - **onRestoreInstanceState(Bundle)**: Restore saved state after recreation.

- **Menu / UI controls**
  - **onCreateOptionsMenu(Menu)**: Initialize menu options.
  - **onOptionsItemSelected(MenuItem)**: Handle menu selections.

- **Context-provided Methods**
  - **getApplicationContext()**: Access app-level context.
  - **getSystemService(String)**: Access system services (e.g., LOCATION_SERVICE).
  - **getSharedPreferences(name, mode)**: App storage for small data.

- **Dialogs, toasts, and interaction**
  - **runOnUiThread(Runnable)**: Update UI from a background thread.
  - **requestPermissions(String[], int)**: Ask for runtime permissions.
  - **onRequestPermissionsResult(...)**: Handle permission results.

3.2 Context

- **What is it**: Context is an interface to global information about your application environment. It gives you access to:
  - App resources (colors, strings, layouts, drawables, etc.)
  - System services (e.g. LayoutInflater, PowerManager, NotificationManager, etc.)
  - Permissions
  - Starting new Activities, Services, etc.

  When you create a View in code:

```
0  Button btn = new Button(this);
1  GridLayout grid = new GridLayout(this);
```

  That this is a Context. In an Activity, this works because Activity is a subclass of Context.

  The View needs a Context to:

  - Know which theme/style to apply
  - Access resources (e.g., strings, colors, dimensions)
  - Hook into the Android system for rendering

  Without a Context, a View has no ”connection” to the running Android app environment.

3.3 ConstraintLayout

- **Needed includes**:

```
import androidx.constraintlayout.widget.ConstraintLayout;
import androidx.constraintlayout.widget.ConstraintSet;
import android.view.ViewGroup;
```

- **Create ConstraintLayout**:

```
ConstraintLayout layout = new ConstraintLayout(this);
```

The argument *this* is the **context**, which in this case is the activity.

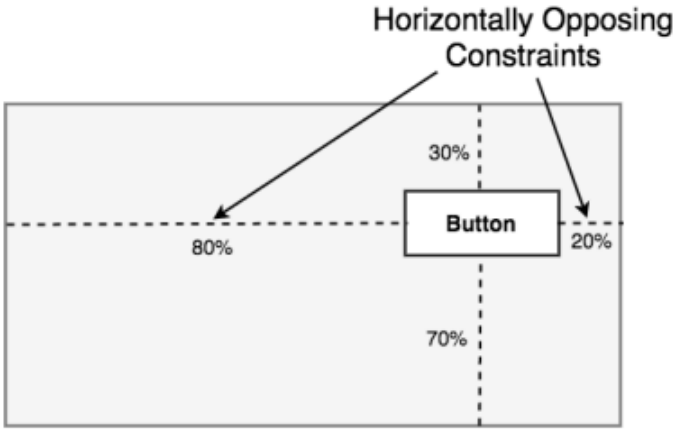
- **Constraints**: Constraints are essentially sets of rules that dictate the way in which a widget is aligned and distanced in relation to other widgets. The sides of the containing ConstraintLayout and special elements are called **guidelines**.

  Constraints also dictate how the user interface layout of an activity will respond to changes in device orientation, or when displayed on devices of differing screen sizes. In order to be adequately configured, a widget must have sufficient constraint connections such that it's position can be resolved by the ConstraintLayout layout engine in both the horizontal and vertical planes.

- **Margins**: A margin is a form of constraint that specifies a fixed distance.

- **Opposing Constraints**: Two constraints operating along the same axis on a single widget are referred to as opposing constraints. In other words, a widget with constraints on both its left and right-hand sides is considered to have horizontally opposing constraints.

  The key point to understand here is that once opposing constraints are implemented on a particular axis, the positioning of the widget is now based on percentages of the overall dimensions of the ConstraintLayout rather than coordinate based.

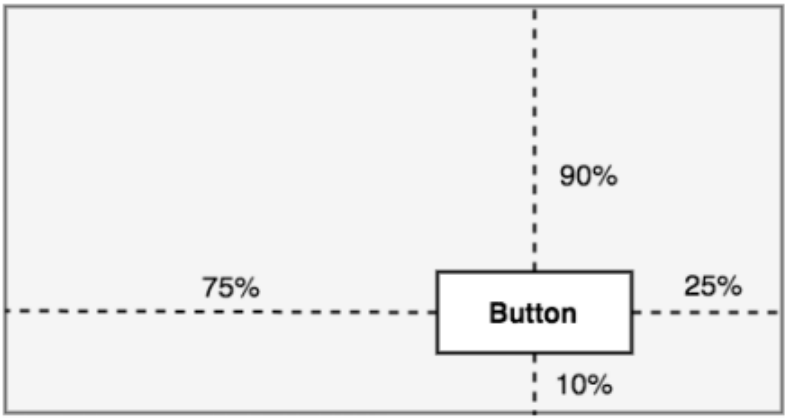


Instead of being fixed at 20dp from the top of the layout, for example, the widget is now positioned at a point 30% from the top of the layout, regardless of the physical dimensions of the container, or parent layout.

In different orientations and when running on larger or smaller screens, the Button will always be in the same location relative to the dimensions of the parent layout.

- **Constraint Bias**: By default, opposing constraints are equal, resulting in the corresponding widget being centered along the axis of opposition.

  To allow for the adjustment of widget position in the case of opposing constraints, the ConstraintLayout implements a feature known as constraint bias. Constraint bias allows the positioning of a widget along the axis of opposition to be biased by a specified percentage in favor of one constraint.



Widget Offset using Constraint Bias

Figure 2b-4

Figure 2b-4, for example, shows the previous constraint layout with a 75% horizontal bias and 10% vertical bias:

- **Chains**: ConstraintLayout chains provide a way for the layout behavior of two or more widgets to be defined as a group.

  Chains can be declared in either the vertical or horizontal axis and configured to define how the widgets in the chain are spaced and sized. Widgets are chained when connected together by bi-directional constraints.

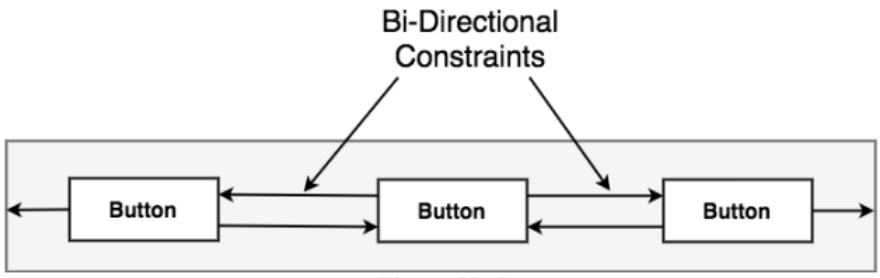


Figure 2b-5

Figure 2b-5, for example, illustrates three widgets chained in this way

The first element in the chain is the chain head which translates to the top widget in a vertical chain or, in the case of a horizontal chain, the left-most widget.

The layout behavior of the entire chain is primarily configured by setting attributes on the chain head widget.

- **Chain Styles**: The layout behavior of a ConstraintLayout chain is dictated by the chain style setting applied to the chain head widget; these are as described next.
  - **Spread Chain**: The widgets contained within the chain are distributed evenly across the available space which is the default behavior for chains.

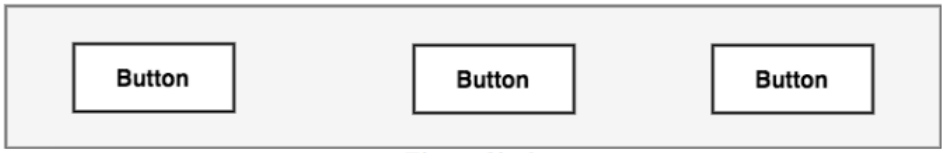


Figure 2b-6

  - **Spread Inside Chain**: The widgets contained within the chain are spread evenly between the chain head and the last widget in the chain. The head and last widgets are not included in the distribution of spacing

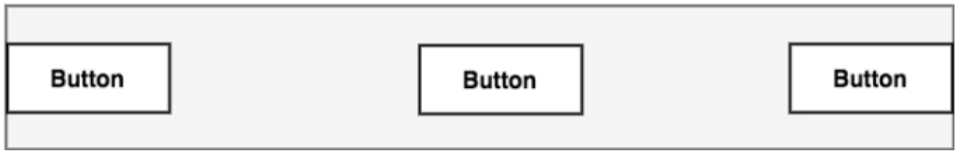


Figure 2b-7

  - **Weighted Chain**: Allows the space taken up by each widget in the chain to be defined via weighting properties.

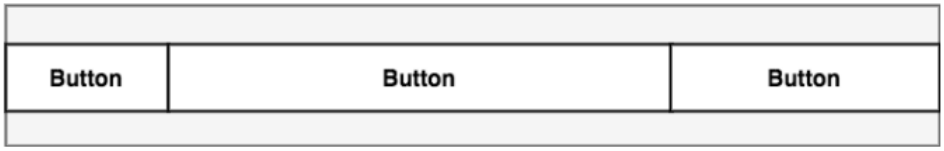


Figure 2b-8

- **Packed Chain**: The widgets that make up the chain are packed together without any spacing. A bias may be applied to control the horizontal or vertical positioning of the chain in relation to the parent container.

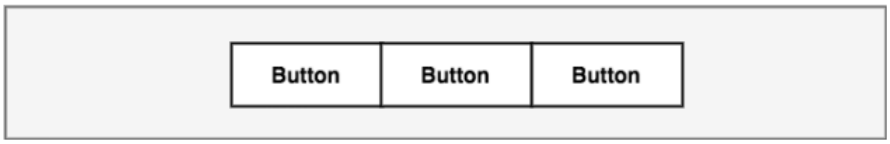


Figure 2b-9

- **Creating chains with java code**: You do it by manually setting constraints directly on each view's ConstraintLayout.LayoutParams, using the leftToRight, rightToLeft, topToBottom, etc. fields.

  Suppose we have three buttons in a ConstraintLayout,

```
// --- Button 1 constraints ---
ConstraintLayout.LayoutParams lp1 = new
    ↪ ConstraintLayout.LayoutParams(
        ViewGroup.LayoutParams.WRAP_CONTENT,
        ViewGroup.LayoutParams.WRAP_CONTENT
);
lp1.leftToLeft = ConstraintLayout.LayoutParams.PARENT_ID;
lp1.rightToLeft = btn2.getId();    // Chain with Button 2
lp1.topToTop = ConstraintLayout.LayoutParams.PARENT_ID;
lp1.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID;
lp1.horizontalChainStyle =
    ↪ ConstraintLayout.LayoutParams.CHAIN_SPREAD;
btn1.setLayoutParams(lp1);

// --- Button 2 constraints ---
ConstraintLayout.LayoutParams lp2 = new
    ↪ ConstraintLayout.LayoutParams(
        ViewGroup.LayoutParams.WRAP_CONTENT,
        ViewGroup.LayoutParams.WRAP_CONTENT
);
lp2.leftToRight = btn1.getId();
lp2.rightToLeft = btn3.getId();
lp2.topToTop = ConstraintLayout.LayoutParams.PARENT_ID;
lp2.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID;
btn2.setLayoutParams(lp2);

// --- Button 3 constraints ---
ConstraintLayout.LayoutParams lp3 = new
    ↪ ConstraintLayout.LayoutParams(
        ViewGroup.LayoutParams.WRAP_CONTENT,
        ViewGroup.LayoutParams.WRAP_CONTENT
);
lp3.leftToRight = btn2.getId();
lp3.rightToRight = ConstraintLayout.LayoutParams.PARENT_ID;
lp3.topToTop = ConstraintLayout.LayoutParams.PARENT_ID;
lp3.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID;
btn3.setLayoutParams(lp3);
```

- **Chain styles in java**:

```
ConstraintLayout.LayoutParams.CHAIN_SPREAD
ConstraintLayout.LayoutParams.CHAIN_SPREAD_INSIDE
ConstraintLayout.LayoutParams.CHAIN_PACKED
```

Notice that in the above example, we give `ConstraintLayout.LayoutParams.CHAIN_SPREAD` to the head button in the chain.

- **Weighted chain in java**: To make a weighted chain, you must:
  1. Use `MATCH_CONSTRAINT` (0dp) for the dimension you want to weight (width in a horizontal chain, height in a vertical one).
  2. Assign a weight to each view using layoutParams.horizontalWeight or layoutParams.verticalWeight.

3. Use either `CHAIN_SPREAD` or `CHAIN_SPREAD_INSIDE` style.

For example, consider again those three buttons

```
// ---- Button 1 ----
ConstraintLayout.LayoutParams lp1 = new
    ↪ ConstraintLayout.LayoutParams(
        0,  // MATCH_CONSTRAINT for weighted width
        ViewGroup.LayoutParams.WRAP_CONTENT
);
lp1.leftToLeft = ConstraintLayout.LayoutParams.PARENT_ID;
lp1.rightToLeft = btn2.getId();
lp1.horizontalWeight = 1f; // weight = 1
lp1.horizontalChainStyle =
    ↪ ConstraintLayout.LayoutParams.CHAIN_SPREAD;
btn1.setLayoutParams(lp1);

// ---- Button 2 ----
ConstraintLayout.LayoutParams lp2 = new
    ↪ ConstraintLayout.LayoutParams(
        0,
        ViewGroup.LayoutParams.WRAP_CONTENT
);
lp2.leftToRight = btn1.getId();
lp2.rightToLeft = btn3.getId();
lp2.horizontalWeight = 2f; // weight = 2
btn2.setLayoutParams(lp2);

// ---- Button 3 ----
ConstraintLayout.LayoutParams lp3 = new
    ↪ ConstraintLayout.LayoutParams(
        0,
        ViewGroup.LayoutParams.WRAP_CONTENT
);
lp3.leftToRight = btn2.getId();
lp3.rightToRight = ConstraintLayout.LayoutParams.PARENT_ID;
lp3.horizontalWeight = 1f; // weight = 1
btn3.setLayoutParams(lp3);
```

- **MATCH_CONSTRAINT**: Let's break down what MATCH_CONSTRAINT means, how it differs from WRAP_CONTENT and MATCH_PARENT, and when you use it.

  MATCH_CONSTRAINT is a special size mode in ConstraintLayout that tells a view:

  "Size yourself dynamically based on your constraints, rather than fixed content or parent size."

  In Java, it's specified by setting the dimension (width or height) to 0 in the layout params:

```
o  ConstraintLayout.LayoutParams params = new
   ↪  ConstraintLayout.LayoutParams(0, WRAP_CONTENT);
```

View size is flexible and determined by the constraints and optionally by weights or ratios

- **Baseline Alignment**: So far, we have only referred to constraints that dictate alignment relative to the sides of a widget (typically referred to as side constraints).

  A common requirement, however, is for a widget to be aligned relative to the content that it displays rather than the boundaries of the widget itself. To address this need, ConstraintLayout provides baseline alignment support.

  Every view that displays text (e.g. TextView, EditText, Button) has a **text baseline** - the imaginary horizontal line upon which the text "sits."

  Baseline alignment means you're aligning two or more views based on that text baseline, instead of their tops or bottoms.

  Suppose we require a TextView widget to be positioned 40dp to the left of the Button.

  In this case, the TextView needs to be baseline aligned with the Button view.

  This means that the text within the Button needs to be vertically aligned with the text within the TextView.

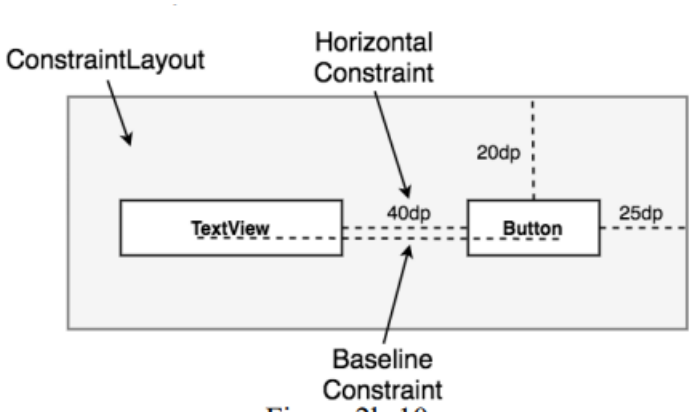


Figure 2b-10

The TextView is now aligned vertically along the baseline of the Button and positioned 40dp horizontally from the Button object's left-hand edge.

In Java, you connect one view's baseline to another view's baseline using:

```
o  params.baselineToBaseline = otherView.getId();
```

You can combine this with other constraints like leftToLeft, topToTop, etc.

- **Working with Guidelines**: Guidelines are special elements available within the ConstraintLayout that provide an additional target to which constraints may be connected.

  Multiple guidelines may be added to a ConstraintLayout instance which may, in turn, be configured in horizontal or vertical orientations.

  Once added, constraint connections may be established from widgets in the layout to the guidelines. This is particularly useful when multiple widgets need to be aligned along an axis.

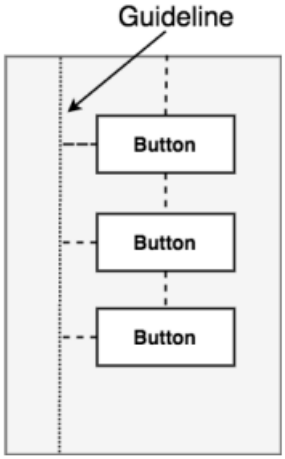


Figure 2b-11

- **Guidelines in java**: First we create a `Guideline` object, then configure `ConstraintLayout.LayoutParams` for it.

```
import androidx.constraintlayout.widget.Guideline

// 1) Make the guideline
Guideline vGuide = new Guideline(this);
vGuide.setId(View.generateViewId());

ConstraintLayout.LayoutParams vgLP = new
↪  ConstraintLayout.LayoutParams(
        ConstraintLayout.LayoutParams.WRAP_CONTENT,
        ConstraintLayout.LayoutParams.WRAP_CONTENT
);
vgLP.orientation = ConstraintLayout.LayoutParams.VERTICAL;
// Choose ONE of these three ways to position it:
vgLP.guidePercent = 0.30f;   // 30% from the left (0..1)
// vgLP.guideBegin  = 120;    // 120px from the left
// vgLP.guideEnd    = 16;     // 16px from the right

vGuide.setLayoutParams(vgLP);
layout.addView(vGuide);

// 2) Constrain a view to the guideline
TextView tv = new TextView(this);
tv.setId(View.generateViewId());
tv.setText("Hello");
ConstraintLayout.LayoutParams tvLP = new
↪  ConstraintLayout.LayoutParams(
0,  // MATCH_CONSTRAINT width so it can expand between
↪  guideline and parent
ConstraintLayout.LayoutParams.WRAP_CONTENT
);
tvLP.leftToRight = vGuide.getId();                        //
↪  to the right of guideline
tvLP.rightToRight = ConstraintLayout.LayoutParams.PARENT_ID;
tvLP.topToTop = ConstraintLayout.LayoutParams.PARENT_ID;

tv.setLayoutParams(tvLP);
layout.addView(tv);

setContentView(layout);
```

Note: `ConstraintLayout.LayoutParams.orientation` is a property that only applies to Guidelines, not to regular Views.

`ConstraintLayout.LayoutParams.orientation` tells the ConstraintLayout whether a Guideline is:

  - **Vertical**: divides the layout left ↔ right
  - **Horizontal**: divides the layout top ↔ bottom

- **Working with Barriers**: Rather like guidelines, barriers are virtual views that can be used to constrain views within a layout

As with guidelines, a barrier can be vertical or horizontal and one or more views may be constrained to it (to avoid confusion, these will be referred to as constrained views).

Unlike guidelines where the guideline remains at a fixed position within the layout, however, the position of a barrier is defined by a set of so called reference views.

Barriers were introduced to address an issue that occurs with some frequency involving overlapping views.

Consider the following example

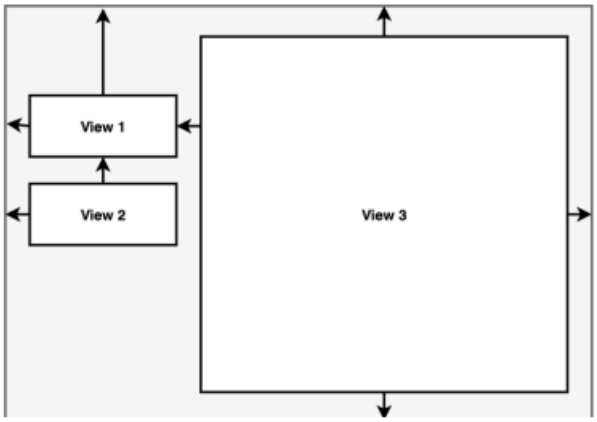


Figure 2b-12

The key points to note about the above layout is that the width of View 3 is set to match constraint mode, and the left-hand edge of the view is connected to the right hand edge of View 1.

As currently implemented, an increase in width of View 1 will have the desired effect of reducing the width of View 3:

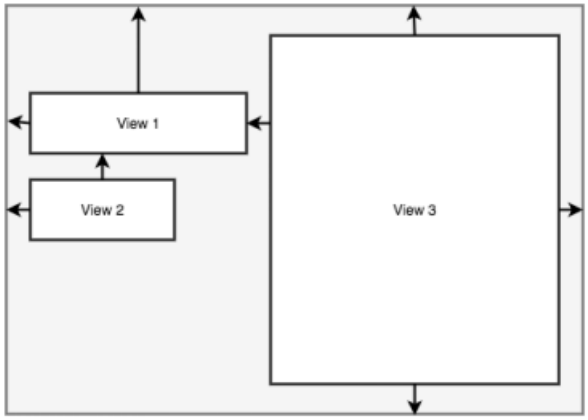


Figure 2b-13

A problem arises, however, if View 2 increases in width instead of View 1:

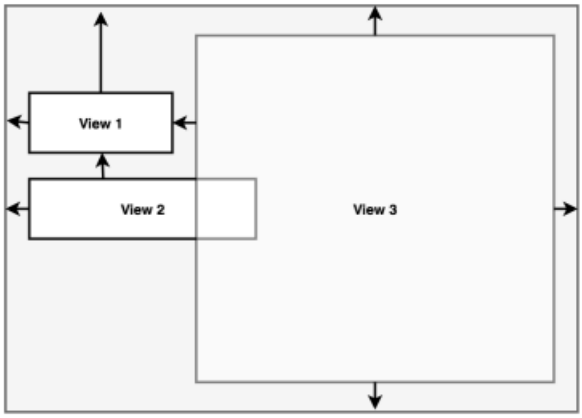


Figure 2b-14

Clearly because View 3 is only constrained by View 1, it does not resize to accommodate the increase in width of View 2 causing the views to overlap.

A solution to this problem is to add a vertical barrier and assign Views 1 and 2 as the barrier's reference views so that they control the barrier position.

The left-hand edge of View 3 will then be constrained in relation to the barrier, making it a constrained view.

Now when either View 1 or View 2 increase in width, the barrier will move to accommodate the widest of the two views, causing the width of View 3 change in relation to the new barrier position:

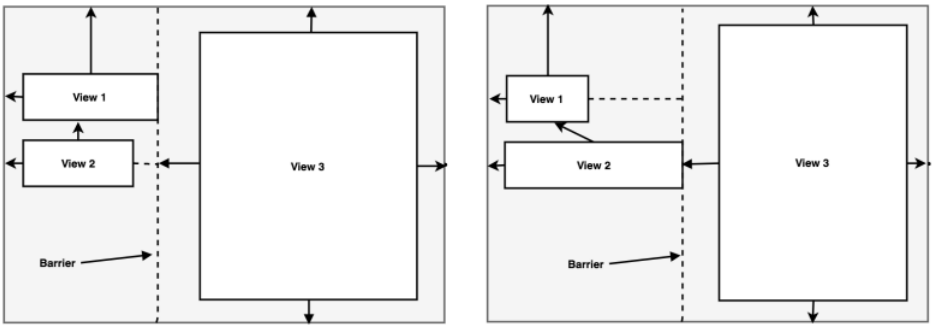


Figure 2b-15

When working with barriers there is no limit to the number of reference views and constrained views that can be associated with a single barrier.

- **Barriers in java**: A Barrier is a virtual helper object in ConstraintLayout that positions itself dynamically based on the position of other views. It's like a movable guideline that automatically adjusts to the furthest edge of a group of views.

Assume we have two TextViews

```
import androidx.constraintlayout.widget.Barrier;

// --- Create a barrier ---
Barrier barrier = new Barrier(this);
barrier.setId(View.generateViewId());
barrier.setType(Barrier.END); // or LEFT, RIGHT, START, TOP, BOTTOM
barrier.setReferencedIds(new int[]{label1.getId(), label2.getId()});
layout.addView(barrier);

// Create a button that stays to the right of both text views
Button btn = new Button(this);
btn.setId(View.generateViewId());
btn.setText("Aligned with longest");
ConstraintLayout.LayoutParams lp3 = new ConstraintLayout.LayoutParams(
        ConstraintLayout.LayoutParams.WRAP_CONTENT,
        ConstraintLayout.LayoutParams.WRAP_CONTENT
);
lp3.startToEnd = barrier.getId(); // use barrier directly
lp3.topToTop = t1.getId();       // align top with first label
lp3.leftMargin = 16;
btn.setLayoutParams(lp3);

layout.addView(btn);

...
```

- **Barrier types**: Barrier types you can use:
  - Barrier.START
  - Barrier.END
  - Barrier.LEFT
  - Barrier.RIGHT
  - Barrier.TOP
  - Barrier.BOTTOM

  Which are explained in the following table

Constant	Tracks the...	Moves toward...	Typical Use
Barrier.START	Start edges (left in LTR, right in RTL)	Start side	Keep something to the left of the leftmost view
Barrier.END End	edges (right in LTR, left in RTL)	End side	Keep something to the right of the rightmost view
Barrier.LEFT	Left edges (absolute)	Left side	Same as START but ignores layout direction
Barrier.RIGHT	Right edges (absolute)	Right side	Same as END but ignores layout direction
Barrier.TOP	Top edges	Upward	Keep something below the topmost/-tallest view
Barrier.BOTTOM	Bottom edges	Downward	Keep something below the lowest/bottommost view

- **ConstraintLayout.LayoutParams**: Has constructor

```
ConstraintLayout.LayoutParams(int width, int height)
```

Lets you define width and height (MATCH_PARENT, WRAP_CONTENT, or px).

For example,

```
ConstraintLayout.LayoutParams lp =
    new ConstraintLayout.LayoutParams(
    ViewGroup.LayoutParams.WRAP_CONTENT,
    ViewGroup.LayoutParams.WRAP_CONTENT
);
```

Then, we can add constraints, Each field takes an ID of another view (or PARENT_ID):

  - **leftToLeft**: Align the left edge of this view to the left edge of another view (or parent).
  - **leftToRight**: Align the left edge of this view to the right edge of another view.
  - **rightToLeft**: Align the right edge of this view to the left edge of another view.
  - **rightToRight**: Align the right edge of this view to the right edge of another view.
  - **topToTop**: Align the top edge of this view to the top edge of another view.

- **topToBottom**: Align the top edge of this view to the bottom edge of another view.
- **bottomToTop**: Align the bottom edge of this view to the top edge of another view.
- **bottomToBottom**: Align the bottom edge of this view to the bottom edge of another view.
- **startToStart**: Align the start edge of this view to the start edge of another view.
- **startToEnd**: Align the start edge of this view to the end edge of another view.
- **endToStart**: Align the end edge of this view to the start edge of another view.
- **endToEnd**: Align the end edge of this view to the end edge of another view.
- **horizontalBias(float b)**: $b \times 100$ percent from left
- **verticalBias(float b)**: $b \times 100$ percent from top

Note:

- left/right = physical edges (always left/right of the screen).
- start/end = logical edges (switch meaning in RTL layouts).

For example,

```
lp.topToTop = ConstraintLayout.LayoutParams.PARENT_ID;
```

We also have chains, for arranging multiple views in a line with flexible spacing:

- **Retrieve and update LayoutParams**:

```
ConstraintLayout.LayoutParams lp =
    (ConstraintLayout.LayoutParams) view.getLayoutParams();
...
view.setLayoutParams(lp);
```

3.4 GridLayout

GridLayout is a type of ViewGroup that arranges its children in a grid of rows and columns.

Each child view is placed into a "cell" defined by its row and column. You can make a child span multiple rows or columns. It's similar to a table layout but more flexible (alignments, spans, etc.).

- **Needed Includes**:

```
import android.content.Context;
import android.view.ViewGroup;
import android.widget.GridLayout;
```

- **Create GridLayout**: You can make a GridLayout in code just like any other layout:

```
GridLayout grid = new GridLayout(this);   // "this" =
↪  Context, usually Activity
grid.setRowCount(3);                      // number of rows
grid.setColumnCount(3);                   // number of
↪  columns
```

- **Setting LayoutParams for GridLayout**: We use ViewGroup.LayoutParams to set the layout params for the entire grid container.

```
grid.setLayoutParams(
    new ViewGroup.LayoutParams(
        ViewGroup.LayoutParams.MATCH_PARENT,
        ViewGroup.LayoutParams.MATCH_PARENT
    )
);
```

- **GridLayout.LayoutParams**: You need GridLayout.LayoutParams for child views placed inside a GridLayout. GridLayout.LayoutParams is the special layout parameter class that children of a GridLayout must use.

  It tells the GridLayout parent:

  - Which row and column this child belongs to
  - How many cells to span
  - How to align inside those cells
  - How much margin space it should have

  For example,

```
Button btn = new Button(this);
btn.setText("Hi");

GridLayout.LayoutParams btnLp = new GridLayout.LayoutParams(
        GridLayout.spec(0),  // row 0
        GridLayout.spec(1)   // column 1
);
btnLp.width = GridLayout.LayoutParams.WRAP_CONTENT;
btnLp.height = GridLayout.LayoutParams.WRAP_CONTENT;

btn.setLayoutParams(btnLp);
grid.addView(btn);
```

`GridLayout.spec(int index)` creates a Spec object. A Spec describes a position in either rows or columns. Here, GridLayout.spec(0) means row 0 (the first row), and GridLayout.spec(1) means the first column.

- **GridLayout.spec overloads**:

```
GridLayout.spec(int index) // Single index, default span=1, default alignment = UNDEFINED
GridLayout.spec(int index, int size) // Index + span
GridLayout.spec(int index, Alignment align) // Index + alignment
GridLayout.spec(int index, int size, Alignment align) // Index + span + alignment
```

You can pass these alignment constants:

  - **GridLayout.START**: align to start (left or top).
  - **GridLayout.END**: align to end (right or bottom).
  - **GridLayout.CENTER**: center inside the cell.
  - **GridLayout.FILL**: expand to fill the cell.

(BASELINE exists for aligning text baselines in rows.)

- **Rule of thumb**: Use ViewGroup.LayoutParams when sizing a container relative to its parent.

  Use the container's own LayoutParams subclass (GridLayout.LayoutParams, LinearLayout.LayoutParams, etc.) when adding children inside that container.

- **Using MarginLayoutParams on GridLayout.setLayoutParams**: This affects the GridLayout as a whole, not its children.

  For example,

```
ViewGroup.MarginLayoutParams lp =
new ViewGroup.MarginLayoutParams(
    ViewGroup.LayoutParams.MATCH_PARENT,
    ViewGroup.LayoutParams.WRAP_CONTENT
);

lp.setMargins(20, 40, 20, 40); // left, top, right, bottom in
↪  px

grid.setLayoutParams(lp);
```

3.5 Button

- **Subclass**: Buttons are a subclass of TextView, which is a subclass of View
- **Creating buttons**: There are basically 3 parts:
  1. Construct the Button (needs a Context)
  2. Customize it (text, size, colors, etc.)
  3. Add it to a parent layout (like GridLayout, LinearLayout, etc.) with proper LayoutParams.

  To create the button,

```
Button myButton = new Button(context);
```

- **Button methods**:
  - setTextSize(int size)
  - setOnClickListener(listener)
  - setEnabled(boolean status)
  - setText(string text)
  - setLayoutParams(lp)

3.6 TextView and EditText

- **Creating TextView**: TextView is a subclass of View

```
TextView textView = new TextView(context);
```

- **TextView methods**:
  - setWidth(int w)
  - setHeight(int h)
  - setGravity(Gravity g)
  - setBackgroundColor(Color c)
  - setTextSize(int size)
  - setText(string text)
  - setBackgroundColor(Color c)
  - setLayoutParams(lp)
  - setTypeFace(Typeface font, int style)
- **Creating EditText**: EditText is a subclass of TextView, which is a subclass of View

```
EditText editText = new EditText(context);
```

- **EditText methods**:
  - setWidth(int w)
  - setHeight(int h)
  - setHint(String)
  - setGravity(Gravity g)
  - setBackgroundColor(Color c)
  - setTextSize(int size)
  - setText(string text)
  - setBackgroundColor(Color c)
  - setLayoutParams(lp)

3.7 AutoCompleteTextView

- **Intro**: AutoCompleteTextView is just an EditText with a dropdown list of suggestions. Those suggestions come from an Adapter (usually ArrayAdapter).
- **XML**

```
<AutoCompleteTextView
android:id="@+id/autoTextView"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:hint="Type something"
android:completionThreshold="1" />
```

`android:completionThreshold="1"` means start showing suggestions after typing 1 character.

- **Java**

```
AutoCompleteTextView autoText =
    ↪ findViewById(R.id.autoTextView);

// List of suggestions
String[] courses = {
    "CSCI 240",
    "CSCI 241",
    "CSCI 330",
    "CSCI 340",
    "MATH 240",
    "MATH 350"
};

// Adapter
ArrayAdapter<String> adapter = new ArrayAdapter<>(
this,
android.R.layout.simple_dropdown_item_1line,
courses
);

autoText.setAdapter(adapter);
```

- **Handling clicks**

```
autoText.setOnItemClickListener(new
    AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View
        view, int position, long id) {
        String selected =
            parent.getItemAtPosition(position).toString();
        Toast.makeText(MainActivity.this, "You chose: " +
            selected, Toast.LENGTH_SHORT).show();
    }
});
```

- **With SQLite DB**

```
ArrayList<String> names = db.getAllNames();  // from SQLite
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
    android.R.layout.simple_dropdown_item_1line, names);
autoText.setAdapter(adapter);
```

3.8 Color

- **Color constants**: This class defines color constants and helper methods for working with colors.

```
Color.RED
Color.BLUE
Color.BLACK
Color.WHITE
```

  These are just integer values representing ARGB colors.

- **Create colors**:
  - ARGB values (alpha, red, green, blue):

```
int myColor = Color.argb(255, 100, 200, 150);
```

    Here, 255 = fully opaque.
  - RGB values (no alpha, alpha = 255):

```
int myColor = Color.rgb(100, 200, 150);
```

  - Parse from string:

```
int myColor = Color.parseColor("#FF5722");  // hex code
```

3.9 Context

3.10 Configuration

3.11 Resources

3.12 Resources.Theme

3.13 TypedValue

- **What is it**: TypedValue is a lightweight container class in the Android framework. It holds the raw, unresolved, low-level data for a resource value after the system loads it from XML, resources, or themes.

  It is used internally whenever Android needs to interpret a resource, and it is used by developers when retrieving resource attributes dynamically (for example, via resolveAttribute, getValue, or theme-based lookups).

- **What TypedValue stores**: A TypedValue instance contains:
  - **type:** the kind of resource (dimension, string, color, integer, reference, attribute, etc.).
  - **data:** the raw value, usually a packed integer containing the actual information.
  - **density:** density information for images or units.
  - **string:** the string content when the resource is a text value.
  - **resourceId:** the ID of the referenced resource, if applicable.

  Note: The important point is that TypedValue does not interpret the data itself; it simply stores what the system loaded, and you then use helper methods to convert or resolve it.

- **Resolving a theme attribute**: This is the most common usage in application code:

```
TypedValue tv = new TypedValue();
if (getTheme().resolveAttribute(android.R.attr.actionBarSize
↪ , tv, true)) {
    int px = TypedValue.complexToDimensionPixelSize(
        tv.data,
        getResources().getDisplayMetrics()
    );
}
```

  - `resolveAttribute` fills the TypedValue with the raw data from the current theme.
  - `complexToDimensionPixelSize` interprets the packed dimension stored in tv.data.

- **Retrieving primitive resource values**:

```
getResources().getValue(R.dimen.padding_large, tv, true);
```

- **Complex data format**: A complex data value is something stored in the binary form defined in TypedValue. It represents:
  - dimensions (e.g., 16dp, 24sp)
  - fractions (e.g., 50%p)
  - other packed values with units

  The internal integer layout looks like:
  - **high 8 bits:** unit type (dp, sp, px, in, mm)

  - **next bits:** radix (how the number was represented)
  - **last 23 bits:** mantissa

- **Why the Data Is "Complex"**: Most of Android's dimensions, colors, and numbers are not stored as simple integers. They are stored in a compact binary form called a complex data format.

  For example,

  - 16dp
  - 14sp
  - FF00FF
  - ?attr/colorPrimary

3.14 DisplayMetrics

3.15 Log

3.16 WindowManager (Interface)

3.17 Display

3.18 Gravity

- **Intro**: This class defines constants used to position or align content inside a View.

  It doesn't move the view itself — it controls how the content inside a view (like text in a TextView) or a child inside a parent layout is aligned.

  - Gravity.LEFT / Gravity.RIGHT
  - Gravity.TOP / Gravity.BOTTOM
  - Gravity.CENTER (both horizontally + vertically)
  - Gravity.CENTER_HORIZONTAL
  - Gravity.CENTER_VERTICAL
  - Gravity.FILL (stretch to fill)

  Use bitwise OR (|) to combine:

```
0  textView.setGravity(Gravity.CENTER | Gravity.BOTTOM);
```

3.19 DiaglogInterface and AlertDialog

- **AlertDialog**: A subclass of Dialog. Used to show a modal pop-up window on top of the activity — typically for alerts, confirmations, or choices It can have:
  - A title
  - A message
  - Optional icon
  - Up to 3 buttons (Positive, Negative, Neutral)
  - Custom layouts (if you want more than just text)
- **Create an alert**:

```
AlertDialog.Builder alert = new AlertDialog.Builder(this);
...
AlertDialog dialog = alert.create();
dialog.show();
```

- **AlertDialog.builder methods**
  - alert.setTitle(string title);
  - alert.setMessage(string message);
  - alert.setPositiveButton(string buttonText, DialogInterface.onClickListener);
  - alert.setNegativeButton(string buttonText, DialogInterface.onClickListener);
  - alert.show();
- **DialogInterface**: This is just an interface. Many dialog-related classes (including AlertDialog) implement it.

```
public interface DialogInterface {
    void cancel();
    void dismiss();

    interface OnCancelListener {
        void onCancel(DialogInterface dialog);
    }

    interface OnDismissListener {
        void onDismiss(DialogInterface dialog);
    }

    interface OnClickListener {
        void onClick(DialogInterface dialog, int which);
    }

    // ... and others like OnKeyListener,
    ↪  OnMultiChoiceClickListener
}
```

It gives you common methods to control the dialog:

- **dismiss()**: close the dialog
- **cancel()**: cancel the dialog (triggers onCancel() callback)

It's also used in listeners for button clicks.

- **DialogInterface.OnClickListener**: We can use this interface to show the alert when something is clicked.

```
0  private class MyDialog implements
   ↪  DialogInterface.OnClickListener
1  {
2
3      public void onClick(DialogInterface dialog, int which)
4      {
5
6      }
7  }
```

The which parameter in listeners

- DialogInterface.BUTTON_POSITIVE (-1)
- DialogInterface.BUTTON_NEGATIVE (-2)
- DialogInterface.BUTTON_NEUTRAL (-3)

So you know which button was pressed.

The dialog parameter is a reference to the dialog that triggered the click. Its type is the interface DialogInterface, but in practice it will usually be an instance of a concrete class like AlertDialog. You can use this reference to control the dialog inside the callback:

- **dialog.dismiss()**: close the dialog immediately.
- **dialog.cancel()**: cancel the dialog (triggers OnCancelListener if one is set).

```
0  builder.setPositiveButton("OK", new
   ↪  DialogInterface.OnClickListener() {
1      @Override
2      public void onClick(DialogInterface dialog, int id) {
3          // dialog is the AlertDialog, typed as
           ↪  DialogInterface
4          dialog.dismiss();   // closes it
5      }
6  });
```

3.20 GradientDrawable

- **Includes**:

```
import android.graphics.drawable.GradientDrawable;
```

- **GradientDrawable**: A GradientDrawable is a drawable object (something you can use as a background or graphic) that can display:
  - Solid colors, or Gradients (color transitions),

  and can have:
  - Rounded corners,
  - Borders (strokes),
  - Different shapes (rectangle, oval, line, ring).

  Essentially, it's Android's built-in shape painter

- **Creation**:

```
GradientDrawable shape = new GradientDrawable();
```

- **Methods**:

Method	Description	Example Usage
`setShape(int shape)`	Sets the type: `RECTANGLE`, `OVAL`, `LINE`, `RING`	`setShape(GradientDrawable.OVAL)`
`setCornerRadius(float radius)`	Rounds the corners (only works for rectangles)	`setCornerRadius(30f)`
`setCornerRadii(float[] radii)`	Gives each corner a different roundness	`setCornerRadii(new float[]20,20,0,0,20,20,0,0)`
`setColor(int color)`	Fills with a solid color	`setColor(Color.BLUE)`
`setStroke(int width, int color)`	Adds a border	`setStroke(3, Color.WHITE)`
`setGradientType(int type)`	Chooses gradient: `LINEAR`, `RADIAL`, `SWEEP`	`setGradientType(GradientDrawable.LINEAR_GRA`
`setColors(int[] colors)`	Defines colors for gradient transitions	`setColors(new int[]Color.RED, Color.YELLOW)`

- **Using GradientDrawable**: We can then call .setBackground() on a view, passing in our GradientDrawable.

- **Giving a button rounded edges**:

```
Button b = new Button();

GradientDrawable button_shape = new GradientDrawable();
button_shape.setShape(GradientDrawable.RECTANGLE);
button_shape.setCornerRadius(35f);
button_shape.setColor(PURPLE);

b.setBackground(button_shape);
```

3.21 android.graphics.Typeface

- **Typeface class**: the Typeface class in Android is the foundation for all text styling related to fonts and weight (bold, italic, etc.). It represents the font face used to render text on screen — in a TextView, Canvas, or anywhere text is drawn.

  A `Typeface` is an object that describes the style and family of a font. It defines how text looks — e.g. whether it's serif or sans-serif, bold or italic, or even a custom font you loaded.

- **Font family constants**:

Constant	Font family	Appearance
Typeface.DEFAULT	Default system font (usually Roboto on newer Androids)	Plain
Typeface.SANS_SERIF	Sans-serif	Clean, modern
Typeface.SERIF	Serif	Classic (like Times New Roman)
Typeface.MONOSPACE	Monospace	Fixed-width (like Courier New)

- **Weight constants**:

Constant	Style
Typeface.NORMAL	0
Typeface.BOLD	1
Typeface.ITALIC	2
Typeface.BOLD_ITALIC	3

- **Using with setTypeFace**

```
label.setTypeface(Typeface.SERIF, Typeface.BOLD);
```

- **Null as a parameter**: We can set the font family but not weight

```
label.setTypeface(null, Typeface.BOLD);
```

  We can set the family but not weight by using the overload that only accepts a family

```
label.setTypeface(Typeface.SERIF);
```

- **Loading custom font from res/font**: If you have `res/font/roboto_bold.ttf`, you can load it like:

```
Typeface roboto = ResourcesCompat.getFont(this,
    ↪ R.font.roboto_bold);
textView.setTypeface(roboto);
```

or programmatically from assets:

```
Typeface tf = Typeface.createFromAsset(getAssets(),
    ↪ "fonts/CustomFont.ttf");
textView.setTypeface(tf);
```

- **Methods**:

Method	Description
`create(Typeface family, int style)`	Returns a new Typeface based on an existing family and style.
`createFromAsset(AssetManager mgr, String path)`:	Loads a Typeface from an asset file (assets/fonts/...).
`createFromFile(File path)`:	Loads from a file on disk.
`defaultFromStyle(int style)`:	Returns the default Typeface for a style (e.g. Typeface.defaultFromStyle(Typeface.BOLD)).
`equals(Object obj)`:	Compares two typefaces.
`hashCode()`:	Hash for comparison.

3.22 Relative layout

- **Include**

```
android.widget.RelativeLayout
```

- **Relative layout**: RelativeLayout is a ViewGroup that lets you position child views relative to each other or to the parent container.

  Each child view can be placed relative to the parent (top, bottom, left, right, center, etc.) or relative to another view (above, below, to the left/right of another widget).

- **Create a RelativeLayout**:

```
RelativeLayout layout = new RelativeLayout(this);
layout.setLayoutParams(new ViewGroup.LayoutParams(
    ViewGroup.LayoutParams.MATCH_PARENT,
    ViewGroup.LayoutParams.MATCH_PARENT
));
```

- **Create RelativeLayout.LayoutParams**

```
RelativeLayout.LayoutParams rlp = new
↪   RelativeLayout.LayoutParams(new ViewGroup.LayoutParams(
    RelativeLayout.LayoutParams.WRAP_CONTENT,
    RelativeLayout.LayoutParams.WRAP_CONTENT
));
```

- **.addRule()**: Adds a layout rule to be interpreted by the RelativeLayout. There are two versions

```
void addRule(int verb, int subject)
void addRule(int verb)
```

  The first version applies a standalone rule - one that does not reference another view.

```
params.addRule(RelativeLayout.CENTER_IN_PARENT);
```

  The rule means: ”center this view both horizontally and vertically inside the parent”

  The second version defines a relationship between this view and another view (by ID).

3.23 Linear layout

- **Creating linear layout with layout params**:

```
LinearLayout root = new LinearLayout(this);
root.setOrientation(LinearLayout.VERTICAL);
root.setPadding(dp(16), dp(16), dp(16), dp(16));
root.setLayoutParams(new LinearLayout.LayoutParams(
        LinearLayout.LayoutParams.MATCH_PARENT,
        LinearLayout.LayoutParams.MATCH_PARENT
));
root.setGravity(Gravity.START);
```

3.24 Table layout and Table row

- **TableLayout**: TableLayout arranges children into rows and columns. But unlike HTML tables, it only positions elements, it does not:
  - Draw borders
  - Automatically size columns evenly
  - Support row/column spanning without special params

  Each row must be a TableRow, and each TableRow contains children like TextView, EditText, Button, etc.

- **Creating TableLayout**

```
TableLayout table = new TableLayout(this);
table.setLayoutParams(new TableLayout.LayoutParams(
        TableLayout.LayoutParams.MATCH_PARENT,
        TableLayout.LayoutParams.MATCH_PARENT
));
table.setStretchAllColumns(true); // Makes columns expand
    ↪ evenly
```

- **Creating rows**:

```
// Create TableRows and Views (cells)
for (int i = 0; i < 3; i++) {
    TableRow row = new TableRow(this);
    row.setLayoutParams(new TableLayout.LayoutParams(
            TableLayout.LayoutParams.MATCH_PARENT,
            TableLayout.LayoutParams.WRAP_CONTENT
    ));

    for (int j = 0; j < 3; j++) {
        TextView cell = new TextView(this);
        cell.setText("R" + i + " C" + j);
        cell.setPadding(dp(8), dp(8), dp(8), dp(8));
        cell.setGravity(Gravity.CENTER);

        row.addView(cell);
    }

    // Add row to table
    table.addView(row);
}
```

3.25 Frame layout

3.26 ListView

- **Intro**: A ListView is a UI component that displays a vertical scrolling list of items. It's an AdapterView that shows one row per item, must use an Adapter to supply the row data

  It does **not** know:

  - how many items you have
  - how each row looks
  - how to convert your data (Strings, objects, etc.) into a row view

  The Adapter handles those things.

- **Example**:

```
ListView listView = findViewById(R.id.listView);

String[] animals = {"Dog", "Cat", "Bird"};

ArrayAdapter<String> adapter = new ArrayAdapter<>(
        this,
        android.R.layout.simple_list_item_1,
        animals
);

listView.setAdapter(adapter);
```

  ArrayAdapter expects the layout to contain a TextView with id @android:id/text1, which is what the built-in android.R.layout.simple_list_item_1 provides

3.27 Adapter, AdapterView, ArrayAdapter, BaseAdapter

- **Adapter**: An Adapter is an object in Android that acts as a bridge between a data source and an AdapterView (such as a ListView, GridView, Spinner, or AutoCompleteTextView).

  Its primary responsibility is to provide access to the data and create views for each data item so the AdapterView can display them on the screen.

  Adapters allow data of any kind—arrays, lists, database results, custom objects—to be converted into uniform, displayable views.

- **Purposes of an Adapter**: The Adapter serves two main purposes:

  1. **Supplies Data**: The Adapter tells the AdapterView how many items exist (`getCount()`), what data is at a given position (`getItem(position)`)
  2. **Supplies Views**: The Adapter is responsible for producing a View that represents each item, usually through converting data into TextViews, or inflating custom row layouts

     It does this through the method:

     ```
     getView(int position, View convertView, ViewGroup
     ↪ parent)
     ```

     This method returns the actual row or grid cell that appears on screen.

- **AdapterView**: An AdapterView is an abstract view in Android that displays a collection of items by binding the view to an Adapter. It provides the framework for presenting data in a structured, scrollable layout, such as a list or a grid.

  The AdapterView itself does not store or manage the data; instead, it relies on an Adapter to supply both the data and the corresponding child views.

- **Purpose of an AdapterView**: The purpose of an AdapterView is to:

  - **Display multiple data items** (e.g., a list of strings, objects, etc.)
  - **Create and manage views** for each item using an Adapter
  - **Handle user interaction** with the individual items (such as clicks)

- **Common AdapterView subclasses**: Examples of views that extend AdapterView include:

  - **ListView:** displays items in a vertically scrollable list
  - **GridView:** displays items in a grid of rows and columns
  - **Spinner:** displays a dropdown list of choices
  - **AutoCompleteTextView:** provides suggestions based on user input

- **ArrayAdapter**: An ArrayAdapter is a concrete subclass of the Adapter class in Android that binds an array or list of objects to an AdapterView, such as a ListView, Spinner, or AutoCompleteTextView. It provides a simple way to convert each element of a Java array or List<T> into a View—typically a single TextView.

- **ArrayAdapter purpose**: The ArrayAdapter is designed to:
  - Store a collection of data items (usually Strings, but can be any object type)
  - Provide the number of items to the AdapterView
  - Create a View for each item, typically using a standard Android layout
- **Example**:

```
String[] colors = {"Red", "Green", "Blue"};
ArrayAdapter<String> adapter = new ArrayAdapter<>(
        this,
        android.R.layout.simple_list_item_1,
        colors
);
listView.setAdapter(adapter);
```

  Where the arguments are

  - **Context**: The Activity or Application context.
  - **Resource ID for the row layout**: Usually a layout with a single TextView, e.g. android.R.layout.simple_list_item_1
  - **Data source**: An array (String[]) or a list (ArrayList<String>)

  Internally, ArrayAdapter's job is to take each data item and convert it into a View.

  For simple layouts:

  - It inflates the row layout (e.g., simple_list_item_1)
  - It finds the TextView
  - It sets the text to the item's string value

  If the data is not a string, it uses the item's toString() method.

  ArrayAdapter works best with layouts that contain one TextView or contain a TextView as the root element

- **Adapter vs ArrayAdapter**: Adapter is an interface / abstract concept in Android.

  It describes:

  - What an adapter must do
  - How it supplies data to an AdapterView

  An Adapter must provide methods such as:

  - **getCount()**: how many items
  - **getItem(position)**: return data at position
  - **getView(position, convertView, parent)**: return a View for the item

  You almost never create an Adapter yourself because it is too general. Instead, you use one of its concrete subclasses, like ArrayAdapter.

  ArrayAdapter is a specific class that implements the Adapter contract. It works with arrays and ArrayLists

3.28 Image view

3.29 Compound Button

3.30 Check box

3.31 RadioGroup and Radio Buttons

3.32 Abs spinner

3.33 Spinner

3.34 Progress bar

3.35 Abs seek bar

3.36 Seek bar

3.37 AttributeSet

3.38 Constraint set

3.39 defStyleAttr

3.40 defStyleRes

3.41 android.content.Intent

- **Intent**: An Intent is an Android messaging object used to:
  - Start a new Activity
  - Start a Service
  - Deliver a broadcast to other apps or system components
  - Request an action from another app (e.g., open browser, camera)

  An Intent is how Android apps request actions, move between screens, and share data.
- **Explicit intent**: You specify the exact component (Activity) you want to start.

```
Intent intent = new Intent(MainActivity.this,
↪  SecondActivity.class);
startActivity(intent);
```

  Start SecondActivity from MainActivity:
- **Implicit intent**: You ask the system to find an app that can handle the requested action, used to interact with other apps.

```
Uri url = Uri.parse("https://www.google.com");
Intent intent = new Intent(Intent.ACTION_VIEW, url);
startActivity(intent);
```

  Here, Android opens whatever browser the user chooses.
- **Passing data with an intent**: To send data to another Activity:

```
Intent intent = new Intent(MainActivity.this,
↪  SecondActivity.class);
intent.putExtra("username", "Nathan");
intent.putExtra("age", 21);
startActivity(intent);
```

  Retrieve it in SecondActivity:

```
String name = getIntent().getStringExtra("username");
int age = getIntent().getIntExtra("age", 0);
```

- **Updating AndroidManifest**: When we add an activity to an app, we need to add a corresponding activity element to the AndroidManifest.xml file

```
<activity
    android:name=".classname"
    android:screenOrientation="portrait">
</activity>
```

3.42 android.view.Display

- **Display class**: The android.view.Display class represents a physical screen or display device that your app's UI can be shown on. It provides detailed information about the screen your app is running on — such as its size, refresh rate, orientation, and pixel density.

  It is useful when adapting layouts, scaling graphics, or handling multi-screen setups

- **Getting a display object**: You usually don't create Display yourself. Instead, you retrieve it from a system service like `WindowManager`

```
WindowManager wm = (WindowManager)
    ↪ getSystemService(Context.WINDOW_SERVICE);
Display display = wm.getDefaultDisplay();
```

  Or, with

```
Display display = getWindowManager().getDefaultDisplay();
```

  or, in newer Android versions (API 30+):

```
Display display = getDisplay();  // available from any
    ↪ Activity or View
```

  If you're in a non-Activity class (like a helper or controller class), you can get it through a Context:

```
WindowManager wm = (WindowManager)
    ↪ context.getSystemService(Context.WINDOW_SERVICE);
Display display = wm.getDefaultDisplay();
```

- **Getting display size**: To get the size of the usable screen in pixels:

```
Display display = getWindowManager().getDefaultDisplay();

Point size = new Point();
display.getSize(size);

int width = size.x;
int height = size.y;
```

- **Getting Real Screen Size**: To include everything (status bar, navigation bar):

```
Point realSize = new Point();
display.getRealSize(realSize);
```

- **Getting Refresh Rate**: Returns how many times per second the screen updates:

```
float refreshRate = display.getRefreshRate();
```

- **Getting Display Rotation**: Tells you how the screen is currently rotated relative to its "natural" orientation:

```
int rotation = display.getRotation();
```

  where the possible values are

  - **Surface.ROTATION_0**: natural orientation
  - **Surface.ROTATION_90**: rotated right
  - **Surface.ROTATION_180**: upside down
  - **Surface.ROTATION_270**: rotated left

- **Getting Display Metrics**: To obtain screen density and scaling info:

```
DisplayMetrics metrics = new DisplayMetrics();
display.getMetrics(metrics);

int densityDpi = metrics.densityDpi;
float density = metrics.density;  // Scale factor for dp → px
float scaledDensity = metrics.scaledDensity;  // Scale for sp
↪  → px
```

- **In Multi-Display or External Display Scenarios**: Starting from Android 4.2+, a device can have multiple displays (like casting to a TV or projector). You can access all of them with:

```
DisplayManager dm = (DisplayManager)
↪  getSystemService(Context.DISPLAY_SERVICE);
Display[] displays = dm.getDisplays();
```

- **Methods**:
  - **getSize(Point)**: Gets the app-usable screen size (in pixels).
  - **getRealSize(Point)**: Gets the full physical display size.
  - **getRotation()**: Returns the screen rotation (0, 90, 180, 270).
  - **getRefreshRate()**: Returns display refresh rate in Hz.
  - **getMetrics(DisplayMetrics)**: Returns logical density and scaling info.
  - **getName()**: Returns display name (useful in multi-display setups).

3.43 KeyEvent

- **KeyEvent**: `android.view.KeyEvent` represents a hardware key press or release event — like when the user presses or releases a key on the device's keyboard, a game controller, or a button such as Volume Up, Back, or Enter.

  KeyEvent objects are delivered to your app whenever a key action happens.

  They describe:

  - Which key was pressed (keyCode)
  - Whether it was a press or release
  - The time, source, modifiers (Shift, Ctrl, etc.)
  - The Unicode character it represents (if any)

- **Lifecycle**

Stage	Event Type	Constant
Key pressed down	ACTION_DOWN	KeyEvent.ACTION_DOWN
Key released	ACTION_UP	KeyEvent.ACTION_UP
Key held (repeats)	multiple ACTION_DOWN events	with getRepeatCount() > 0

- **Getting a KeyEvent object**: In Android, you don't manually create KeyEvent objects in most cases. Instead, the Android framework automatically provides them to your app when a user presses or releases a hardware or software key.

  When a key is pressed, the system calls your Activity, View, or Dialog methods and passes a KeyEvent object as a parameter.

  - In an Activity:

```
0  @Override
1  public boolean onKeyDown(int keyCode, KeyEvent event) {
2      Log.d("KeyEvent", "Pressed key: " +
       ↪  event.getKeyCode());
3      return super.onKeyDown(keyCode, event);
4  }
5
6  @Override
7  public boolean onKeyUp(int keyCode, KeyEvent event) {
8      Log.d("KeyEvent", "Released: " +
       ↪  event.getKeyCode());
9      return true;
10 }
```

    Here, the system creates and passes the KeyEvent object automatically.

  - In a View:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    // Handle key press inside your custom view
    return true;
}
```

Or use a listener:

```
view.setOnKeyListener(new View.OnKeyListener() {
    @Override
    public boolean onKey(View v, int keyCode, KeyEvent
    ↪ event) {
        if (event.getAction() == KeyEvent.ACTION_DOWN) {
            Log.d("KeyEvent", "Key pressed: " +
            ↪ event.getKeyCode());
            return true;
        }
        return false;
    }
});
```

Note: onKeyDown() and onKeyUp() return a boolean because the return value tells the Android framework whether your code has consumed (handled) the key event or no

When you return true, it means "I've handled this key event — don't send it anywhere else."

Returning false means: "I didn't handle this — let the system or another component handle it." Then Android passes the event along:

- From the current View up to its parent
- From the Activity to the Window
- Or eventually to the system (for default behavior)

super.onKeyDown() calls the default handler in the base Activity class, which performs standard Android behaviors (like handling BACK or MENU keys).

- **Getting the action**

```
MotionEvent event.getAction()
```

- **Action constants**:
  - **ACTION_DOWN**: Key was pressed down
  - **ACTION_UP**: Key was released
  - **ACTION_MULTIPLE**: Multiple repeated key events (e.g., long press)

For example,

```
if (event.getAction() == KeyEvent.ACTION_DOWN) { ... }
```

- **Getting the key code**:

```
int event.getKeyCode();
```

- **Key Code Constants**: These tell you which key was pressed.

  There are hundreds of these - a few common groups:

```
KEYCODE_A, KEYCODE_B, ..., KEYCODE_Z
KEYCODE_0, KEYCODE_1, ..., KEYCODE_9
```

```
KEYCODE_ENTER
KEYCODE_DEL           // Backspace
KEYCODE_TAB
KEYCODE_ESCAPE
KEYCODE_SPACE
KEYCODE_BACK
KEYCODE_MENU
KEYCODE_HOME
```

```
KEYCODE_VOLUME_UP
KEYCODE_VOLUME_DOWN
KEYCODE_MUTE
KEYCODE_MEDIA_PLAY_PAUSE
KEYCODE_MEDIA_NEXT
KEYCODE_MEDIA_PREVIOUS
```

```
KEYCODE_DPAD_UP
KEYCODE_DPAD_DOWN
KEYCODE_DPAD_LEFT
KEYCODE_DPAD_RIGHT
KEYCODE_BUTTON_A
KEYCODE_BUTTON_B
```

```
KEYCODE_POWER
KEYCODE_SLEEP
KEYCODE_WAKEUP
```

- **Checking if shift was pressed**:

```
boolean event.isShiftPressed()
```

- **Getting meta state**

```
event.getMetaState()
```

- **Meta / modifier key flags**: Used for Shift, Alt, Ctrl, etc. These can be combined using bitwise OR (|).
  - **META_SHIFT_ON**: Shift key active
  - **META_ALT_ON**: Alt key active
  - **META_CTRL_ON**: Control key active
  - **META_META_ON**: Meta/Command key active
  - **META_SYM_ON**: Symbol modifier active
  - **META_CAPS_LOCK_ON**: Caps lock active
  - **META_NUM_LOCK_ON**: Num lock active
  - **META_SCROLL_LOCK_ON**: Scroll lock active
- **KeyEvent Methods**
  - **getAction()**: int Down, Up, or Multiple
  - **getKeyCode()**: int Which key was pressed
  - **getMetaState()**: int Modifier flags
  - **getRepeatCount()**: int How many times repeated
  - **getEventTime()**: long Time when event occurred
  - **getDownTime()**: long Time when key was first pressed
  - **getDeviceId()**: int ID of the input device (keyboard/gamepad)
  - **getScanCode()**: int Raw hardware scan code
  - **getUnicodeChar()**: int Unicode value (e.g., 'A' → 65)
  - **getFlags()**: int Internal system flags
  - **getSource()**: int Input source (keyboard, gamepad, etc.)
  - **isShiftPressed()**: boolean True if Shift active
  - **isCtrlPressed()**: boolean True if Ctrl active
  - **isAltPressed()**: boolean True if Alt active
- **Other useful constants**
  - **Action Constants**: ACTION_DOWN, ACTION_UP
  - **Key Codes**: KEYCODE_A, KEYCODE_ENTER, KEYCODE_BACK
  - **Meta Flags**: META_SHIFT_ON, META_CTRL_ON
  - **Event Data Members**: getAction(), getKeyCode(), getDownTime(), getRepeatCount(), etc.
  - **Flags**: FLAG_LONG_PRESS, FLAG_SOFT_KEYBOARD, etc.

3.44 Animations

- **Create AnimationSet**:

```
AnimationSet animation = new AnimationSet(boolean
↪  shareInterpolator)
```

An Interpolator controls the animation speed pattern over time:

- **true**: (default) All animations inside AnimationSet use the same interpolator (the one set on the AnimationSet itself)
- **false**: Each animation can define its own interpolator

```
AnimationSet set = new AnimationSet(true); // true = share
↪  interpolator

// Fade-in animation
AlphaAnimation alpha = new AlphaAnimation(0f, 1f);
alpha.setDuration(1000);

// Move-up animation
TranslateAnimation move = new TranslateAnimation(
0, 0,
50f, 0f
);
move.setDuration(1000);

// Add animations to set
set.addAnimation(alpha);
set.addAnimation(move);

// Start the animation
view.startAnimation(set);
```

3.45 SharedPreferences, SharedPreferences.Editor, and PreferencesManager

- **Get a SharedPreferences object**

```
SharedPreferences prefs = getSharedPreferences("MyPrefs",
↪  MODE_PRIVATE);
```

  - **"MyPrefs"**: the filename to store in
  - **MODE_PRIVATE**: → only your app can access it

- **Write / save data**:

```
SharedPreferences.Editor editor = prefs.edit();
editor.putString("username", "Nate");
editor.putBoolean("isDarkMode", true);
editor.apply(); // async (recommended)
```

  - **.apply()**: saves in the background
  - **.commit()**: saves immediately but blocks the thread — only use if you must know the result instantly

- **Read data**

```
String username = prefs.getString("username", "Guest");
boolean darkMode = prefs.getBoolean("isDarkMode", false);
```

  The second argument is the default value if the key does not exist

- **Remove data**

```
prefs.edit().remove("username").apply();
```

- **Wipe everything**

```
prefs.edit().clear().apply();
```

- **Where is the data stored**: SharedPreferences saves to:

```
/data/data/<your package name>/shared_prefs/MyPrefs.xml
```

3.46 Menu and MenuItem

- **Which menu type?**: We use the Menu and MenuItem class to create an **Options Menu**.
- **Creating an options menu**: We override `onCreateOptionsMenu()`. Android automatically calls this when the Activity starts. Here, you'll use the Menu object (passed as a parameter) to add items manually.

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // groupId, itemId, order, title
    menu.add(0, 101, 0, "Settings");
    menu.add(0, 102, 1, "Help");
    menu.add(0, 103, 2, "Exit");

    // You can also configure each item after adding it
    MenuItem settingsItem = menu.findItem(101);
    settingsItem.setIcon(android.R.drawable.ic_menu_preferen⌋
    ↪ ces);
    settingsItem.setShowAsAction(MenuItem.SHOW_AS_ACTION_IF_⌋
    ↪ ROOM);

    return true;  // tells Android to display this menu
}
```

menu is the Menu interface object, menu.add(...) returns a MenuItem object. You can customize each MenuItem afterward.

- **Handle clicks**: When the user taps a menu item, Android passes a MenuItem object representing what was clicked.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case 101:
        Toast.makeText(this, "Settings clicked",
        ↪ Toast.LENGTH_SHORT).show();
        return true;
        case 102:
        Toast.makeText(this, "Help clicked",
        ↪ Toast.LENGTH_SHORT).show();
        return true;
        case 103:
        finish(); // close app
        return true;
        default:
        return super.onOptionsItemSelected(item);
    }
}
```

3.47 SubMenu

- **What is a SubMenu**: A SubMenu is basically a menu nested inside another menu item. SubMenu is an interface that extends Menu, you don't instantiate it directly — instead, you get it from a Menu object using addSubMenu()

- **Example, creating SubMenu**:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Create a regular menu item
    menu.add(0, 1, 0, "Settings");

    // Create a submenu under "File"
    SubMenu fileSubMenu = menu.addSubMenu("File");

    // Add items to that submenu
    fileSubMenu.add(0, 2, 0, "New");
    fileSubMenu.add(0, 3, 1, "Open");
    fileSubMenu.add(0, 4, 2, "Save");

    // Another normal menu item
    menu.add(0, 5, 3, "Help");

    return true;
}
```

- **Handle clicks**: You handle clicks the same way as normal menu items:

```java
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case 1:
            Toast.makeText(this, "Settings clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
        case 2:
            Toast.makeText(this, "New clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
        case 3:
            Toast.makeText(this, "Open clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
        case 4:
            Toast.makeText(this, "Save clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
        case 5:
            Toast.makeText(this, "Help clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
    }
    return super.onOptionsItemSelected(item);
}
```

- **SubMenus in XML**: Example,

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/andr
↪  oid">
    <item android:title="File">
        <menu>
            <item android:id="@+id/action_new"
            ↪  android:title="New" />
            <item android:id="@+id/action_open"
            ↪  android:title="Open" />
            <item android:id="@+id/action_save"
            ↪  android:title="Save" />
        </menu>
    </item>
    <item android:id="@+id/action_help" android:title="Help"
    ↪  />
</menu>
```

Same structure; you just nest a <menu> inside an <item>.

3.48 ContextMenu

- **What is it**: Context menus appear when you long-press a View — like text, a button, or a list item. They're used for actions specific to that item, not the whole Activity.
- **Register a view for a Context Menu**: We use `registerForContextMenu(view v)`

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    TextView myTextView = findViewById(R.id.myTextView);

    // Tell Android this view should show a context menu on
    ↪  long-press
    registerForContextMenu(myTextView);
}
```

- **Override onCreateContextMenu**: This method is called automatically when the user long-presses the registered view. Here you use the Menu interface to build the menu in code.

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenu.ContextMenuInfo
                                ↪  menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);

    // You can set a header or title
    menu.setHeaderTitle("Choose an action");

    // Add items programmatically
    menu.add(0, 101, 0, "Edit");
    menu.add(0, 102, 1, "Share");
    menu.add(0, 103, 2, "Delete");
}
```

  - **ContextMenu Menu**: This is the menu object that you fill with items.
  - **View v**: The view that was long pressed
  - **ContextMenu.ContextMenuInfo menuInfo**: Provides extra context information about the view that was pressed. It's often null unless the view supports structured data — e.g., a ListView or RecyclerView.

    For lists, it contains which row was long-pressed:

  You call super to allow the parent class (AppCompatActivity) to perform its own setup logic before or after you modify the menu.

- **Handle clicks**:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case 101:
            Toast.makeText(this, "Edit clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
        case 102:
            Toast.makeText(this, "Share clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
        case 103:
            Toast.makeText(this, "Delete clicked",
            ↪  Toast.LENGTH_SHORT).show();
            return true;
        default:
            return super.onContextItemSelected(item);
    }
}
```

3.49 PopupMenu

- **What is it**: A PopupMenu is a small floating menu that appears anchored to a specific View — for example, when you tap a button with "⋮" or "More options." It's a temporary dropdown menu used for quick actions — not tied to the Action Bar or long-press events.

- **Example**:

```
Button button = findViewById(R.id.myButton);

button.setOnClickListener(v -> {
    // Create the popup menu, anchored to the button
    PopupMenu popup = new PopupMenu(MainActivity.this, v);

    // Get the Menu object inside the popup
    Menu menu = popup.getMenu();

    // Add items manually
    menu.add(0, 101, 0, "Edit");
    menu.add(0, 102, 1, "Share");
    menu.add(0, 103, 2, "Delete");

    // Handle clicks on the popup items
    popup.setOnMenuItemClickListener(item -> {
        switch (item.getItemId()) {
            case 101:
                Toast.makeText(MainActivity.this, "Edit
                ↪  clicked", Toast.LENGTH_SHORT).show();
                return true;
            case 102:
                Toast.makeText(MainActivity.this, "Share
                ↪  clicked", Toast.LENGTH_SHORT).show();
                return true;
            case 103:
                Toast.makeText(MainActivity.this, "Delete
                ↪  clicked", Toast.LENGTH_SHORT).show();
                return true;
            default:
                return false;
        }
    });

    // Finally, show the popup
    popup.show();
});
```

3.50 Toast

- **What is it**: A Toast is a small popup message that briefly appears at the bottom (or top/center) of the screen. It automatically disappears after a short time.
- **Basic syntax**:

```
Toast.makeText(context, "Hello, world!",
    ↪ Toast.LENGTH_SHORT).show();
```

  - **context**: Usually this, or getApplicationContext() — tells Android which app is showing the Toast
  - **Toast.LENGTH_SHORT**: How long it shows (LENGTH_SHORT ≈ 2s, LENGTH_LONG ≈3.5s)
  - **.show()**: Displays the Toast

- **Duration options**
  - **Toast.LENGTH_SHORT**: ∼2 seconds
  - **Toast.LENGTH_LONG**: ∼3.5 seconds
- **Changing position**: You can move them using setGravity():

```
Toast toast = Toast.makeText(this, "Top toast!",
    ↪ Toast.LENGTH_SHORT);
toast.setGravity(Gravity.TOP | Gravity.CENTER_HORIZONTAL, 0,
    ↪ 200);
toast.show();
```

  The last two numbers (xOffset, yOffset) adjust the offset in pixels

- **Custom Layout Toast**: You can even use a custom XML layout instead of plain text:

```
LayoutInflater inflater = getLayoutInflater();
View layout = inflater.inflate(R.layout.custom_toast,
    ↪ findViewById(R.id.toastRoot));

Toast toast = new Toast(getApplicationContext());
toast.setDuration(Toast.LENGTH_LONG);
toast.setView(layout);
toast.show();
```

  And res/layout/custom_toast.xml might look like:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/
    res/android"
    android:id="@+id/toastRoot"
    android:background="#AA000000"
    android:padding="10dp"
    android:orientation="horizontal">
    <ImageView
        android:src="@android:drawable/ic_dialog_info" />
    <TextView
        android:text="Custom Toast"
        android:textColor="#fff"
        android:paddingStart="10dp"/>
</LinearLayout>
```

3.51 LayoutInflator

- **What is it**: LayoutInflater is a class that converts XML layout files (.xml) into actual View objects in memory — that your app can display or interact with.

  Your XML layout is like a blueprint for a house. LayoutInflater is the builder that reads that blueprint and constructs the actual house (View hierarchy) in Java.

  Suppose you have a layout file: `res/layout/my_custom_view.xml`

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:padding="10dp">
    <ImageView android:src="@android:drawable/ic_menu_info_details"/>
    <TextView android:text="Hello LayoutInflater!"
        android:paddingStart="8dp"/>
</LinearLayout>
```

  You can load it into memory like this:

```
LayoutInflater inflater = getLayoutInflater();   // or LayoutInflater.from(context)
View view = inflater.inflate(R.layout.my_custom_view, null);
```

  Now view is a fully constructed View object tree — a LinearLayout containing an ImageView and TextView.

  You can then:

  - Add it dynamically to another layout
  - Use it in a custom Toast
  - Return it from an adapter (for example, in a ListView)

- **inflate()**

```
View inflate(int resource, ViewGroup root, boolean attachToRoot)
```

  Where

  - **resource**: The XML layout resource to inflate (e.g., R.layout.my_view)
  - **root**: Optional parent layout to attach to
  - **attachToRoot**: Whether to attach the inflated layout to root immediately

3.52 ScrollView

- **What is it**: A ScrollView is a layout container that provides vertical scrolling for its single child view.

  If your content doesn't fit on the screen (too tall), wrapping it inside a ScrollView lets the user scroll up and down to see the rest.

- **Key Rules**:
  - **Only one direct child**: A ScrollView can host only one child view. If you need multiple items, put them inside a container like LinearLayout or ConstraintLayout inside the ScrollView.
  - **Vertical only**: ScrollView scrolls vertically. For horizontal scrolling, use HorizontalScrollView.
  - **The child must be taller than the screen**: Otherwise, it won't scroll because everything fits on screen already.

- **Example**:

```
ScrollView scrollView = new ScrollView(this);
LinearLayout layout = new LinearLayout(this);
layout.setOrientation(LinearLayout.VERTICAL);

// Add some child views
for (int i = 1; i <= 20; i++) {
    TextView tv = new TextView(this);
    tv.setText("Item " + i);
    layout.addView(tv);
}

// Add layout inside scrollview
scrollView.addView(layout);

// Set as activity content
setContentView(scrollView);
```

- **Scrolling programmatically**

```
scrollView.fullScroll(View.FOCUS_DOWN); // scroll to bottom
scrollView.fullScroll(View.FOCUS_UP);   // scroll to top
scrollView.scrollTo(0, 500);            // scroll to specific
↪  Y position
scrollView.smoothScrollBy(0, 100);      // scroll smoothly
```

- **Common XML attributes**
  - **android:fillViewport="true"**: Forces the child view to expand to fill the screen height (even if content is short).
  - **android:scrollbars="none"**: Hide scrollbars.
  - **android:fadeScrollbars="false"**: Keep scrollbars always visible.
  - **android:overScrollMode="never"**: Disable the "stretch" overscroll glow effect.

3.53 InputEvent

- **What is it**: InputEvent is the abstract base class for all low-level input events in Android. It represents any event generated by an input device.
- **Characteristics**
  - It is not instantiated directly.
  - It provides a unified parent type for different categories of input events.
  - Motion-related and key-related events both derive from it.
- **Two concrete subclasses**:
  1. **MotionEvent**: For touch, hover, scroll, and pointer movements.
  2. **KeyEvent**: For hardware key presses (keyboard, volume keys, etc.).
- **What InputEvent provides**: A unique event ID (getDeviceId()), methods for input device querying (e.g., source type, device capabilities), and generic event metadata.

3.54 MotionEvent

- **What is it**: MotionEvent represents all pointer-based activity on the screen. This includes touch, drag, swipe, pinch, hover, and multi-touch events.
- **What MotionEvent contains**:
  - **Action type**: Indicates what the user did. Constants like `ACTION_DOWN`, `ACTION_UP`, `ACTION_MOVES`, etc.
  - **Coordinates**: The event provides the touch coordinates:
    - **getRawX() / getRawY()**: Screen coordinates
    - **getX() / getY()**: Coordinates relative to the view receiving the event
- **Getting MotionEvents**: In Android, you never create a MotionEvent manually. The system generates it and passes it into specific callback methods.
  - **onTouch(View v, MotionEvent event)**: Used with View.OnTouchListener.
  - **onTouchEvent(MotionEvent event)**: Every View has this method. Override it when you want the view itself to process touches.
  - **dispatchTouchEvent(MotionEvent event)**: Called before onTouchEvent. Used rarely, but can intercept events at the view level.

3.55 GestureDetector

- **Gestures**: A gesture is a pattern of touch movement performed by the user on the screen that the system recognizes as a meaningful interaction. It is **not** just a raw touch event.

  A gesture is a higher-level interpretation built from multiple underlying MotionEvents.

  Formally, a gesture is a sequence of touch actions—such as DOWN, MOVE, and UP—that together form a recognizable interaction pattern such as:

  - Tap
  - Double tap
  - Long press
  - Scroll (drag)
  - Fling (fast swipe)
  - Pinch/zoom (multi-touch)

- **MotionEvent vs Gesture**: A MotionEvent is just made up of

  - Raw data: coordinates, finger ID, action code
  - Low-level
  - One event at a time
  - Examples: ACTION_DOWN, ACTION_MOVE, ACTION_UP

  Whereas a Gesture is

  - A meaningful pattern formed from many MotionEvents
  - High-level
  - Recognized by timing, distance, velocity
  - Examples: “single tap”, “fling”, “scroll”, “double tap”

- **Single tap, scroll, fling**: A single tap gesture usually consists of:

  1. ACTION_DOWN
  2. Short time interval
  3. ACTION_UP
  4. Minimal movement

  A scroll gesture consists of:

  - ACTION_DOWN
  - Many ACTION_MOVE events
  - ACTION_UP

  A fling gesture consists of:

  - ACTION_DOWN
  - ACTION_MOVE events
  - ACTION_UP with high velocity

- **What is it**: GestureDetector is an Android framework class that analyzes a stream of MotionEvent objects and determines whether they form a gesture, such as:

  - Single tap
  - Double tap
  - Long press
  - Scroll
  - Fling (quick swipe)

  It removes the burden of manually interpreting touch sequences. In other words, you feed MotionEvents into GestureDetector, and it calls back into your code with high-level gesture events

- **Why GestureDetector**: Raw MotionEvents give you very low-level data:
  - Coordinates
  - Pressure
  - Pointer index
  - Timestamps
  - Action codes (DOWN, MOVE, UP)

  However, recognizing a gesture like "double tap" or "fling" requires:
  - Timing analysis
  - Movement thresholds
  - Velocity computation
  - Distinguishing tap vs scroll
  - Multi-event patterns

  GestureDetector handles all of this internally.

- **How it works**: There are three components:
  1. **A listener object**: You implement one or both listener interfaces (GestureDetector.OnGestureListener (gestures)) (GestureDetector.OnDoubleTapListener (taps)). These contain multiple callback methods.
  2. **A GestureDetector instance**: You create it in your Activity or View:

     ```
     gestureDetector = new GestureDetector(this, listener);
     ```

  3. **Feeding MotionEvents**: Inside your Activity's onTouchEvent() or a View's touch listener, you pass the MotionEvent to GestureDetector:

     ```
     @Override
     public boolean onTouchEvent(MotionEvent event) {
         return gestureDetector.onTouchEvent(event);
     }
     ```

  After that, GestureDetector examines the movement and automatically decides which callback to trigger.

- **GestureDetector.onTouchEvent(MotionEvent event)**: GestureDetector.onTouchEvent(MotionEvent event) is the core method you call to feed touch input into a GestureDetector. It is the entry point that allows the GestureDetector to examine all MotionEvents and decide whether they form a gesture (tap, scroll, fling, long-press, double-tap, etc.).

Its purpose is to analyze a MotionEvent and determine whether it represents a gesture. You call this method from your Activity or View whenever you receive MotionEvents.

```
0  @Override
1  public boolean onTouchEvent(MotionEvent event) {
2      return gestureDetector.onTouchEvent(event);
3  }
```

This passes the event into the detector.

When you call this method,

1. GestureDetector reads the event type
2. It updates its internal gesture state
3. It checks whether the event sequence matches a known gesture, if so, it calls the appropriate callback on your listener:

3.56 GestureDetector.SimpleOnGestureListener

- **What is it**: GestureDetector.SimpleOnGestureListener is a convenience class that implements both:
  - GestureDetector.OnGestureListener
  - GestureDetector.OnDoubleTapListener

  but provides empty default method implementations. Its purpose is to allow you to override only the gesture callbacks you need, rather than being forced to implement all required methods from the interfaces.

  For example,

```
GestureDetector detector = new GestureDetector(
    context,
    new GestureDetector.SimpleOnGestureListener() {
        @Override
        public boolean onDoubleTap(MotionEvent e) {
            Log.d("GESTURE", "Double tap detected");
            return true;
        }

        @Override
        public boolean onSingleTapConfirmed(MotionEvent e) {
            Log.d("GESTURE", "Single tap confirmed");
            return true;
        }
    }
);
```

  You then forward touch events:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    return detector.onTouchEvent(event);
}
```

  This is the most common real-world usage of gesture detection in Android.

- **Available methods**: Because it implements both gesture interfaces, you may selectively override any of the following:
  - onDown(MotionEvent e)
  - onShowPress(MotionEvent e)
  - onSingleTapUp(MotionEvent e)
  - onSingleTapConfirmed(MotionEvent e)
  - onDoubleTap(MotionEvent e)
  - onDoubleTapEvent(MotionEvent e)
  - onLongPress(MotionEvent e)

- onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY)
- onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)

All of these methods return default values (usually false) in the base class, so you can override only what you need.

3.57 view.MeasureSpec

- **What is it**: A MeasureSpec is a compact instruction that a parent ViewGroup gives to a child View during the measurement phase of layout. It tells the child:
  - How much space it may or must use, and
  - How strictly that space must be followed.

  It is the core mechanism that makes Android's layout negotiation work.

  A MeasureSpec is a 32-bit packed integer that encodes two pieces of information:
  - **A mode**: Describes how the parent wants the child to size itself.
  - **A size**: The pixel value associated with the constraint.

  Together

  $$\text{MeasureSpec} = (\text{mode} << 30) \mid \text{size}.$$

  This is why MeasureSpec is passed around as a plain int.

- **Modes**: These modes define the constraint type.
  1. **EXACTLY**: "You must be exactly this size."

     Produced when:
     - The child uses match_parent
     - The child uses a fixed value like 100dp

     The child must report that exact dimension in measurement.
  2. **AT_MOST**: "You can be any size you want, up to this maximum."

     Produced mainly when:
     - The child uses wrap_content
     - The parent can offer some remaining space (e.g. inside LinearLayout, FrameLayout)

     Child measures its intrinsic content but clamps to the max.
  3. **UNSPECIFIED**: "There is no upper bound. Do whatever your content requires."

     Occurs in special cases:
     - A ScrollView measuring its child
     - Internal platform commands
     - Custom views that need unrestricted measurement

- **How MeasureSpecs Are Used**: During layout:
  1. Parent receives its own MeasureSpecs from its parent.
  2. Parent computes new MeasureSpecs for each child.
  3. Parent calls:

     ```
     child.measure(widthMeasureSpec, heightMeasureSpec);
     ```

  4. Child's onMeasure() reads the MeasureSpec:

```
int mode = MeasureSpec.getMode(widthSpec);
int size = MeasureSpec.getSize(widthSpec);
```

5. Child computes its measured width and height using:

```
setMeasuredDimension(measuredWidth, measuredHeight);
```

The child's measured size must obey the constraints defined by the MeasureSpecs.

- **view.measure**: When we call

```
view.measure(widthMeasureSpec, heightMeasureSpec);
```

calling measure() with MeasureSpecs does change internal measurement state of the View, but it does not change layout, draw, or the final displayed size. It only updates the measured dimensions. After the call to measure,

1. Android executes onMeasure() for that view.
2. The view recalculates its measured width/height.
3. The view stores those values internally by calling:

```
setMeasuredDimension(w, h);
```

So, after the manual measure, these properties are updated:

- getMeasuredWidth()
- getMeasuredHeight()

Those values are now different if the specs forced different sizing. That is the only guaranteed state change.

onMeasure()...

- Interprets the MeasureSpec constraints.
- Decide how large the view wants to be.
- Compute the final measured width/height.

onMeasure() must call setMeasuredDimension() before it finishes.

Styling widgets with java

4.1 View

- **void setBackgroundColor(int color)**: Fills the view's background with a solid color.
- **void setBackground(Drawable background)**: Sets a Drawable as the background.
- **Drawable getBackground()**: Returns the current background drawable.
- **void setForeground(Drawable foreground)**: Draws a Drawable on top of the view's content.
- **void setPadding(int left, int top, int right, int bottom)**: Sets the padding inside the view.
- **int getPaddingLeft() / getPaddingTop() / getPaddingRight() / getPaddingBottom()**: Returns padding values.
- **void setElevation(float elevation)**: Adds shadow depth to the view for visual layering.
- **void setClipToOutline(boolean clipToOutline)**: Clips the view's drawing to its outline (e.g., rounded corners).
- **void setOutlineProvider(ViewOutlineProvider provider)**: Defines the outline shape for shadows and clipping.
- **void setBackgroundTintList(ColorStateList tint)**: Applies tint coloring to the background.
- **void setBackgroundTintMode(PorterDuff.Mode mode)**: Defines how the background tint blends with the original color.
- **void setForegroundTintList(ColorStateList tint)**: Applies tint coloring to the foreground.
- **void setOutlineSpotShadowColor(int color)**: Sets the color of the view's spot shadow.
- **void setOutlineAmbientShadowColor(int color)**: Sets the color of the view's ambient shadow.
- **void setRotationX(float rotationX) / setRotationY(float rotationY)**: Rotates the view around the X or Y axis.
- **void setCameraDistance(float distance)**: Adjusts the 3D perspective depth for rotation effects.

4.2 TextView

- **Color, size, typeface**
  - **void setTextColor(int color)**: Sets the text color.
  - **void setTextColor(ColorStateList colors)**: Sets text colors for different states.
  - **void setHighlightColor(int color)**: Sets selection highlight color.
  - **void setLinkTextColor(int color)**: Sets link color.
  - **void setLinkTextColor(ColorStateList colors)**: Sets link colors for states.
  - **void setTextSize(float size)**: Sets text size in scaled pixels.
  - **void setTextSize(int unit, float size)**: Sets text size with unit.
  - **void setTextScaleX(float size)**: Horizontal text scale.
  - **void setTypeface(Typeface tf)**: Sets typeface.
  - **void setTypeface(Typeface tf, int style)**: Sets typeface and style.
  - **void setAllCaps(boolean allCaps)**: Transforms input to ALL CAPS display.
  - **void setTextAppearance(int resId)**: Applies a text appearance style.
  - **void setTextAppearance(Context context, int resId)**: *Deprecated* in API 23.
- **Auto-size text**
  - **void setAutoSizeTextTypeWithDefaults(int autoSizeTextType)**: Enables default auto-size.
  - **void setAutoSizeTextTypeUniformWithConfiguration(int min, int max, int step, int unit)**: Uniform auto-size config.
  - **void setAutoSizeTextTypeUniformWithPresetSizes(int[] presetSizes, int unit)**: Uniform auto-size with presets.
- **Typography, wrapping, justification**
  - **void setLetterSpacing(float letterSpacing)**: Sets letter spacing.
  - **void setLineSpacing(float add, float mult)**: Extra and multiplier.
  - **void setLineHeight(int lineHeight)**: Explicit line height (px).
  - **void setLineHeight(int unit, float lineHeight)**: Explicit line height with unit.
  - **void setEllipsize(TextUtils.TruncateAt where)**: Ellipsize strategy.
  - **void setBreakStrategy(int breakStrategy)**: Paragraph line-break strategy.
  - **void setLineBreakStyle(int lineBreakStyle)**: Line-break style.
  - **void setLineBreakWordStyle(int lineBreakWordStyle)**: Word-break style.
  - **void setHyphenationFrequency(int hyphenationFrequency)**: Hyphenation setting.
  - **void setJustificationMode(int justificationMode)**: Text justification.
  - **void setFallbackLineSpacing(boolean enabled)**: Respect fallback font metrics.
  - **void setIncludeFontPadding(boolean includepad)**: Include extra ascent/descent padding.
  - **void setFirstBaselineToTopHeight(int px)**: Align first baseline to top padding.

- **void setLastBaselineToBottomHeight(int px)**: Align last baseline to bottom padding.
- **void setLocalePreferredLineHeightForMinimumUsed(boolean flag)**: Locale-preferred min line height.

- **Compound drawables and tints**
  - **void setCompoundDrawables(Drawable left, Drawable top, Drawable right, Drawable bottom)**: L/T/R/B drawables.
  - **void setCompoundDrawablesWithIntrinsicBounds(Drawable left, Drawable top, Drawable right, Drawable bottom)**: With intrinsic bounds.
  - **void setCompoundDrawablesWithIntrinsicBounds(int left, int top, int right, int bottom)**: By resource IDs.
  - **void setCompoundDrawablesRelative(Drawable start, Drawable top, Drawable end, Drawable bottom)**: Start/End variants.
  - **void setCompoundDrawablesRelativeWithIntrinsicBounds(Drawable start, Drawable top, Drawable end, Drawable bottom)**: With intrinsic bounds.
  - **void setCompoundDrawablesRelativeWithIntrinsicBounds(int start, int top, int end, int bottom)**: By resource IDs.
  - **void setCompoundDrawablePadding(int pad)**: Space between text and drawables.
  - **void setCompoundDrawableTintList(ColorStateList tint)**: Drawable tint list.
  - **void setCompoundDrawableTintMode(PorterDuff.Mode mode)**: Porter-Duff tint mode.
  - **void setCompoundDrawableTintBlendMode(BlendMode blendMode)**: BlendMode tinting.

- **Hint & cursor/selection visuals**
  - **void setHint(CharSequence hint)**: Hint text.
  - **void setHint(int resid)**: Hint from resource.
  - **void setHintTextColor(int color)**: Hint color.
  - **void setHintTextColor(ColorStateList colors)**: Hint color states.
  - **void setTextCursorDrawable(Drawable d)**: Cursor drawable.
  - **void setTextCursorDrawable(int resId)**: Cursor drawable by resource.
  - **void setTextSelectHandle(int resId)**: Selection handle (resource).
  - **void setTextSelectHandle(Drawable d)**: Selection handle.
  - **void setTextSelectHandleLeft(int resId)**: Left handle (resource).
  - **void setTextSelectHandleLeft(Drawable d)**: Left handle.
  - **void setTextSelectHandleRight(int resId)**: Right handle (resource).
  - **void setTextSelectHandleRight(Drawable d)**: Right handle.

- **Shadows, transforms, and paint flags**
  - **void setShadowLayer(float radius, float dx, float dy, int color)**: Text shadow.
  - **final void setTransformationMethod(TransformationMethod method)**: Visual text transformation (e.g., password).

  - **void setPaintFlags(int flags)**: Underline/strike-through, etc.
  - **void setElegantTextHeight(boolean elegant)**: Use elegant height metrics.
  - **void setFontFeatureSettings(String settings)**: OpenType features.
  - **boolean setFontVariationSettings(String settings)**: Font variations (axes).
- **Alignment / layout-affecting (often part of style guides)**
  - **void setGravity(int gravity)**: Horizontal/vertical alignment within the view.
  - **void setEms(int ems)**: Exact width in ems (typographic sizing).
  - **void setLines(int lines)**: Exact number of lines (tight control of layout look).
- **Styling-related getters (useful to read current style)**
  - **int getCurrentTextColor()**, **ColorStateList getTextColors()**, **int getHighlightColor()**, **ColorStateList getHintTextColors()**, **ColorStateList getLinkTextColors()**
  - **float getTextSize()**, **Typeface getTypeface()**, **float getLetterSpacing()**, **int getLineHeight()**
  - **TextUtils.TruncateAt getEllipsize()**, **int getBreakStrategy()**, **int getHyphenationFrequency()**, **int getJustificationMode()**
  - **String getFontFeatureSettings()**, **String getFontVariationSettings()**
  - **Drawable[] getCompoundDrawables()**, **Drawable[] getCompoundDrawablesRelative()**, **int getCompoundDrawablePadding()**, **ColorStateList getCompoundDrawableTintList()**, **PorterDuff.Mode getCompoundDrawableTintMode()**, **BlendMode getCompoundDrawableTintBlendMode()**
- **Advanced styling hooks (from the *other* list)**
  - **void drawableStateChanged()**: React to state changes that affect drawables/tints.
  - **int[] onCreateDrawableState(int extraSpace)**: Customize drawable state for styling.
  - **void onDraw(Canvas canvas)**: Custom rendering of text/effects.

4.3 EditText (Use TextView methods)

- **boolean isStyleShortcutEnabled()**: Returns true if style shortcuts (e.g., `Ctrl+B` for bold) are enabled.
- **void setStyleShortcutsEnabled(boolean enabled)**: Enables or disables style shortcuts such as `Ctrl+B`, `Ctrl+I`, etc.
- **void setEllipsize(TextUtils.TruncateAt ellipsis)**: Specifies how overflowing text should be ellipsized (e.g., at the end or middle) instead of wrapped.
- **void setText(CharSequence text, TextView.BufferType type)**: Sets the text and determines how it's stored (e.g., as plain, styled, or editable text), affecting styling.

4.4 Button (Use TextView methods)

4.5 ListView

- **Drawable getDivider()**: Returns the drawable that is drawn between each list item.
- **int getDividerHeight()**: Returns the height of the divider between list items.
- **Drawable getOverscrollFooter()**: Returns the drawable drawn below all list content during overscroll.
- **Drawable getOverscrollHeader()**: Returns the drawable drawn above all list content during overscroll.
- **boolean areFooterDividersEnabled()**: Returns whether footer dividers are currently enabled.
- **boolean areHeaderDividersEnabled()**: Returns whether header dividers are currently enabled.
- **void setCacheColorHint(int color)**: Sets a hint color indicating the solid background behind the list, improving appearance on transparent backgrounds.
- **void setDivider(Drawable divider)**: Sets the drawable that will be drawn between list items.
- **void setDividerHeight(int height)**: Sets the height of the divider drawn between list items.
- **void setFooterDividersEnabled(boolean footerDividersEnabled)**: Enables or disables drawing dividers for footer views.
- **void setHeaderDividersEnabled(boolean headerDividersEnabled)**: Enables or disables drawing dividers for header views.
- **void setOverscrollFooter(Drawable footer)**: Sets the drawable to be drawn below all list content during overscroll.
- **void setOverscrollHeader(Drawable header)**: Sets the drawable to be drawn above all list content during overscroll.
- **void dispatchDraw(Canvas canvas)**: (Protected) Called to draw all child views — can be overridden for custom list-item rendering effects.
- **boolean drawChild(Canvas canvas, View child, long drawingTime)**: (Protected) Draws a single list item onto the canvas; used to customize per-item drawing style.

4.6 ImageView

- **void animateTransform(Matrix matrix)**: Applies a temporary transformation matrix for animation or visual effects.
- **final void clearColorFilter()**: Removes any color filter or tint applied to the image.
- **ColorFilter getColorFilter()**: Returns the active color filter used for tinting or blending.
- **int getImageAlpha()**: Returns the current alpha (transparency) level of the image.
- **Matrix getImageMatrix()**: Returns the current transformation matrix applied to the image.
- **BlendMode getImageTintBlendMode()**: Returns the blending mode used for applying tints.
- **ColorStateList getImageTintList()**: Returns the color tint list applied to the image drawable.
- **PorterDuff.Mode getImageTintMode()**: Returns the blending mode used for applying the tint.
- **ImageView.ScaleType getScaleType()**: Returns the scale type that defines how the image fits within the view bounds.
- **void setAlpha(int alpha)**: *Deprecated in API 16.* Sets the overall opacity of the view (use `setImageAlpha(int)` instead).
- **final void setColorFilter(int color, PorterDuff.Mode mode)**: Applies a tint color and blending mode to the image.
- **void setColorFilter(ColorFilter cf)**: Applies a custom color filter to modify image appearance.
- **final void setColorFilter(int color)**: Applies a tint color to the image using the default blending mode.
- **void setCropToPadding(boolean cropToPadding)**: Determines whether the image is cropped to the view's padding.
- **void setImageAlpha(int alpha)**: Sets the transparency level for the image drawable.
- **void setImageBitmap(Bitmap bm)**: Sets a bitmap as the content of the ImageView.
- **void setImageDrawable(Drawable drawable)**: Sets a drawable as the image content.
- **void setImageIcon(Icon icon)**: Sets an icon as the content.
- **void setImageMatrix(Matrix matrix)**: Applies a transformation matrix to the image drawable.
- **void setImageResource(int resId)**: Sets an image resource by its resource ID.
- **void setImageTintBlendMode(BlendMode blendMode)**: Sets how the tint blends with the image.

- **void setImageTintList(ColorStateList tint)**: Applies a color tint list to the image drawable.
- **void setImageTintMode(PorterDuff.Mode tintMode)**: Defines the blending mode for the tint.
- **void setScaleType(ImageView.ScaleType scaleType)**: Defines how the image should be scaled or cropped to fit within the view.
- **void onDraw(Canvas canvas)**: (Protected) Allows custom drawing of the image—useful for advanced visual effects.

4.7 CompoundButton

- **Drawable getButtonDrawable()**: Returns the drawable used as the button image (e.g., checkmark or radio indicator).
- **BlendMode getButtonTintBlendMode()**: Returns the blending mode used to apply the tint to the button drawable.
- **ColorStateList getButtonTintList()**: Returns the color tint list applied to the button drawable.
- **PorterDuff.Mode getButtonTintMode()**: Returns the blending mode used for tinting.
- **void setButtonDrawable(int resId)**: Sets a drawable resource as the button image.
- **void setButtonDrawable(Drawable drawable)**: Sets a drawable object as the button image.
- **void setButtonIcon(Icon icon)**: Sets an icon as the visual button image.
- **void setButtonTintBlendMode(BlendMode tintMode)**: Defines how the tint color blends with the drawable.
- **void setButtonTintList(ColorStateList tint)**: Applies a tint list (different colors for different states) to the button image.
- **void setButtonTintMode(PorterDuff.Mode tintMode)**: Specifies the tint blending mode.
- **void drawableStateChanged()**: (Protected) Called when the state of the view changes in a way that affects drawable appearance (e.g., pressed, focused, checked).
- **int[] onCreateDrawableState(int extraSpace)**: (Protected) Generates the drawable state array — affects how state-based drawables (like selectors) are drawn.
- **void onDraw(Canvas canvas)**: (Protected) Used for custom drawing — allows overriding default appearance.
- **boolean verifyDrawable(Drawable who)**: (Protected) Ensures the drawable being displayed belongs to this view; relevant for custom visual drawables.

4.8 CheckBox (Use Button and CompoundButton styles)

4.9 RadioButton (Use button and CompoundButton styles)

4.10 Spinner

- **int getDropDownHorizontalOffset()**: Returns the horizontal offset in pixels for positioning the dropdown popup.
- **int getDropDownVerticalOffset()**: Returns the vertical offset in pixels for positioning the dropdown popup.
- **int getDropDownWidth()**: Returns the configured width of the dropdown popup window.
- **int getGravity()**: Returns how the selected item view is positioned within the spinner (e.g., left, center, right).
- **Drawable getPopupBackground()**: Returns the background drawable used for the spinner's dropdown popup.
- **void setDropDownHorizontalOffset(int pixels)**: Sets the horizontal offset in pixels for the spinner's dropdown popup.
- **void setDropDownVerticalOffset(int pixels)**: Sets the vertical offset in pixels for the spinner's dropdown popup.
- **void setDropDownWidth(int pixels)**: Sets the width in pixels for the spinner's dropdown popup window.
- **void setGravity(int gravity)**: Defines how the selected item is positioned within the spinner (e.g., `Gravity.CENTER`).
- **void setPopupBackgroundDrawable(Drawable background)**: Sets a drawable as the background for the dropdown popup.
- **void setPopupBackgroundResource(int resId)**: Sets a background resource for the dropdown popup.
- **CharSequence getPrompt()**: Returns the prompt text displayed when the dialog version of the spinner is shown.
- **void setPrompt(CharSequence prompt)**: Sets custom prompt text for the spinner dialog.
- **void setPromptId(int promptId)**: Sets the prompt text by its string resource ID.
- **int getBaseline()**: Returns the text baseline offset (useful when aligning the spinner with other text views visually).
- **void onLayout(boolean changed, int l, int t, int r, int b)**: (Protected) Handles positioning of spinner items — can affect visual alignment.
- **void onMeasure(int widthMeasureSpec, int heightMeasureSpec)**: (Protected) Determines size and layout — affects how the spinner and dropdown appear.

Styles from AdapterView

- **void setSelection(int position)**: Visually highlights the selected item.
- **void setEmptyView(View emptyView)**: Specifies a view to display when the adapter is empty.

4.11 ProgessBar

- **void drawableHotspotChanged(float x, float y)**: Called when the view hotspot changes and must be propagated to drawables or child views.
- **Drawable getCurrentDrawable()**: Returns the drawable currently used to draw the progress bar.
- **Drawable getIndeterminateDrawable()**: Returns the drawable used to draw the progress bar in indeterminate mode.
- **BlendMode getIndeterminateTintBlendMode()**: Returns the blending mode used to apply the tint to the indeterminate drawable.
- **ColorStateList getIndeterminateTintList()**: Returns the color tint list applied to the indeterminate drawable.
- **PorterDuff.Mode getIndeterminateTintMode()**: Returns the blending mode used to apply the tint to the indeterminate drawable.
- **Interpolator getInterpolator()**: Gets the acceleration curve for the indeterminate animation.
- **BlendMode getProgressBackgroundTintBlendMode()**: Returns the blending mode used to apply the tint to the progress background.
- **ColorStateList getProgressBackgroundTintList()**: Returns the tint list applied to the progress background.
- **PorterDuff.Mode getProgressBackgroundTintMode()**: Returns the blending mode used to apply the tint to the progress background.
- **Drawable getProgressDrawable()**: Returns the drawable used to draw the progress bar in progress mode.
- **BlendMode getProgressTintBlendMode()**: Returns the blending mode used to apply the tint to the progress drawable.
- **ColorStateList getProgressTintList()**: Returns the color tint list applied to the progress drawable.
- **PorterDuff.Mode getProgressTintMode()**: Returns the blending mode used to apply the tint to the progress drawable.
- **BlendMode getSecondaryProgressTintBlendMode()**: Returns the blending mode used to apply the tint to the secondary progress drawable.
- **ColorStateList getSecondaryProgressTintList()**: Returns the color tint list applied to the secondary progress drawable.
- **PorterDuff.Mode getSecondaryProgressTintMode()**: Returns the blending mode used to apply the tint to the secondary progress drawable.
- **void invalidateDrawable(Drawable dr)**: Invalidates the specified drawable, forcing a redraw.
- **void jumpDrawablesToCurrentState()**: Jumps all drawables to their current state instantly, skipping animations.
- **void setIndeterminateDrawable(Drawable d)**: Defines the drawable used to draw the progress bar in indeterminate mode.

- **void setIndeterminateDrawableTiled(Drawable d)**: Defines a tileable drawable for the indeterminate mode.
- **void setIndeterminateTintBlendMode(BlendMode blendMode)**: Specifies the blending mode for applying the indeterminate tint.
- **void setIndeterminateTintList(ColorStateList tint)**: Applies a color tint to the indeterminate drawable.
- **void setIndeterminateTintMode(PorterDuff.Mode tintMode)**: Specifies the blending mode for the indeterminate tint.
- **void setInterpolator(Interpolator interpolator)**: Sets the acceleration curve for the indeterminate animation.
- **void setProgressBackgroundTintBlendMode(BlendMode blendMode)**: Specifies the blending mode for the progress background tint.
- **void setProgressBackgroundTintList(ColorStateList tint)**: Applies a tint to the progress background.
- **void setProgressBackgroundTintMode(PorterDuff.Mode tintMode)**: Specifies the blending mode for the progress background tint.
- **void setProgressDrawable(Drawable d)**: Defines the drawable used to draw the progress bar in progress mode.
- **void setProgressDrawableTiled(Drawable d)**: Defines a tileable drawable for the progress bar in progress mode.
- **void setProgressTintBlendMode(BlendMode blendMode)**: Specifies the blending mode for the progress indicator tint.
- **void setProgressTintList(ColorStateList tint)**: Applies a tint to the progress indicator.
- **void setProgressTintMode(PorterDuff.Mode tintMode)**: Specifies the blending mode for the progress indicator tint.
- **void setSecondaryProgressTintBlendMode(BlendMode blendMode)**: Specifies the blending mode for the secondary progress tint.
- **void setSecondaryProgressTintList(ColorStateList tint)**: Applies a tint to the secondary progress indicator.
- **void setSecondaryProgressTintMode(PorterDuff.Mode tintMode)**: Specifies the blending mode for the secondary progress tint.
- **void drawableStateChanged()**: Called whenever the state of the view changes in a way that affects drawable appearance.
- **void onDraw(Canvas canvas)**: Used to perform custom drawing for the progress bar.
- **boolean verifyDrawable(Drawable who)**: Returns true if the specified drawable is managed and displayed by this view.

4.12 AbsSeekBar

- **void drawableHotspotChanged(float x, float y)**: Called whenever the view hotspot changes and must be propagated to drawables or child views.
- **boolean getSplitTrack()**: Returns whether the track is visually split by the thumb.
- **Drawable getThumb()**: Returns the drawable representing the thumb — the draggable component indicating progress.
- **int getThumbOffset()**: Returns the amount by which the thumb extends beyond the track.
- **BlendMode getThumbTintBlendMode()**: Returns the blending mode used to apply the tint to the thumb drawable.
- **ColorStateList getThumbTintList()**: Returns the tint color list applied to the thumb drawable.
- **PorterDuff.Mode getThumbTintMode()**: Returns the blending mode used to apply the tint to the thumb drawable.
- **Drawable getTickMark()**: Returns the drawable used as the tick mark for each progress position.
- **BlendMode getTickMarkTintBlendMode()**: Returns the blending mode used to apply tint to the tick mark drawable.
- **ColorStateList getTickMarkTintList()**: Returns the tint color list applied to the tick mark drawable.
- **PorterDuff.Mode getTickMarkTintMode()**: Returns the blending mode used to apply tint to the tick mark drawable.
- **void jumpDrawablesToCurrentState()**: Immediately updates all drawables associated with this view to their current state.
- **void setSplitTrack(boolean splitTrack)**: Specifies whether the track should be visually split by the thumb.
- **void setThumb(Drawable thumb)**: Sets the drawable used as the thumb in the progress meter.
- **void setThumbOffset(int thumbOffset)**: Sets the offset allowing the thumb to extend beyond the track visually.
- **void setThumbTintBlendMode(BlendMode blendMode)**: Defines the blending mode used when applying tint to the thumb drawable.
- **void setThumbTintList(ColorStateList tint)**: Applies a tint color list to the thumb drawable.
- **void setThumbTintMode(PorterDuff.Mode tintMode)**: Specifies the blending mode used with the thumb tint.
- **void setTickMark(Drawable tickMark)**: Sets the drawable used as tick marks at each progress position.
- **void setTickMarkTintBlendMode(BlendMode blendMode)**: Specifies the blending mode used to apply tint to the tick mark drawable.

- **void setTickMarkTintList(ColorStateList tint)**: Applies a tint color list to the tick mark drawable.
- **void setTickMarkTintMode(PorterDuff.Mode tintMode)**: Specifies the blending mode used with the tick mark tint.
- **void drawableStateChanged()**: Called whenever the state of the view changes in a way that affects drawable appearance.
- **void onDraw(Canvas canvas)**: Implement this method to perform custom drawing operations for the view.
- **boolean verifyDrawable(Drawable who)**: Returns true if the specified drawable is being displayed by this view; subclasses override when managing custom drawables.

4.13 SeekBar (Use styles from ProgessBar and AbsSeekBar)

Used Methods, constants, and fields

5.1 Activity

- **Methods**:
  - **setContentView(int view)**
  - **Intent getIntent()**: Returns the intent that started this activity.
  - **void startActivity(Intent)**: Starts an activity from an Intent object.
  - **WindowManager getWindowManager()**: Retrieve the window manager for showing custom windows.

5.2 View

- **Methods**
  - **setVisibility(int) / getVisibility()**: Show, hide (INVISIBLE), or remove (GONE) a view.
  - **findViewById(int)**: Get a view inside another view or layout.
  - **getId() / setId(int)**: Get or assign a unique view ID.
  - **setOnClickListener(...)**: Handle click actions.
  - **setEnabled(boolean) / isEnabled()**: Enable/disable a view.
  - **setAlpha(float)**: Adjust transparency.
  - **getX(), getY()**: View's position relative to its parent.
  - **setX(), setY()**: Move a view.
  - **void measure(int widthMeasureSpec, int heightMeasureSpec)**: Determines the measured size of the view.
  - **final int getMeasuredHeight()**: Like getMeasuredHeightAndState(), but only returns the raw height component (that is the result is masked by MEASURED$_SIZE_MASK$).
  - **final int getMeasuredWidth()**: Like getMeasuredWidthAndState(), but only returns the raw width component (that is the result is masked by MEASURED$_SIZE_MASK$).
  - **getWidth(), getHeight()**: View's measured width/height.
  - **setLayoutParams(ViewGroup.LayoutParams)**: Change width, height, margins programmatically.
  - **getLayoutParams()**: Access current layout parameters.
  - **requestLayout()**: Ask parent to re-measure and re-layout (after size change).
  - **void invalidate()**: Redraws the view on screen.
  - **setTranslationX/Y(float)**: Move view without changing layout.
  - **setScaleX/Y(float)**: Scale the view.
  - **setRotation(float)**: Rotate the view.
  - **public void setBackground (Drawable background)**: Set the background to a given Drawable, or remove the background
  - **setBackgroundColor(int color)**: Set the background to a given Drawable, or remove the background
  - **public void setBackgroundResource (int resid)**: Set the background to a given resource. The resource should refer to a Drawable object or 0 to remove the background.
  - **public void setPadding (int left, int top, int right, int bottom)**: Sets the padding.
  - **public void setTextAlignment (int textAlignment)**:
  - **public void setTextDirection (int textDirection)**:
- **Constants**
  - **int TEXT_ALIGNMENT_CENTER**: Center paragraph alignment.
  - **int TEXT_ALIGNMENT_GRAVITY**: Default for root view (gravity).
  - **int TEXT_ALIGNMENT_INHERIT**: Inherit text alignment.

- **int TEXT_ALIGNMENT_TEXT_END**: Align to paragraph end.
- **int TEXT_ALIGNMENT_TEXT_START**: Align to paragraph start.
- **int TEXT_ALIGNMENT_VIEW_END**: Align to view end (RTL-aware).
- **int TEXT_ALIGNMENT_VIEW_START**: Align to view start (RTL-aware).
- **int TEXT_DIRECTION_ANY_RTL**: Any-RTL algorithm.
- **int TEXT_DIRECTION_FIRST_STRONG**: First-strong algorithm.
- **int TEXT_DIRECTION_FIRST_STRONG_LTR**: First-strong, force LTR.
- **int TEXT_DIRECTION_FIRST_STRONG_RTL**: First-strong, force RTL.
- **int TEXT_DIRECTION_INHERIT**: Inherit text direction.
- **int TEXT_DIRECTION_LOCALE**: From system locale.
- **int TEXT_DIRECTION_LTR**: Force LTR.
- **int TEXT_DIRECTION_RTL**: Force RTL.

5.3 view.MeasureSpec

- **Methods**
- **Constants**

5.4 ViewGroup

- **Methods**
  - **addView(View) / addView(View, int) / addView(View, LayoutParams)**: Add children quickly.
  - **removeView(View) / removeViewAt(int) / removeAllViews()**: Remove children.
  - **removeAllViewsInLayout()**: Called by a ViewGroup subclass to remove child views from itself
  - **getChildCount(), getChildAt(int), indexOfChild(View)**: Inspect/manage children.
  - **void bringChildToFront(View child)**: Moves a child to the top of the Z-order.
  - **final void layout(int l, int t, int r, int b)**: Assigns size/position to this view and descendants.

5.5 ViewGroup.LayoutParams

- **Fields**
  - **public int height**: Information about how tall the view wants to be.
  - **public int width**: Information about how wide the view wants to be.
- **Constants**
  - **int FILL_PARENT**: Special value for the height or width requested by a View.
  - **int MATCH_PARENT**: Special value for the height or width requested by a View.
  - **int WRAP_CONTENT**: Special value for the height or width requested by a View.

5.6 ViewGroup.MarginLayoutParams

- **XML**
  - **android:layout_marginTop**
  - **android:layout_marginBottom**
  - **android:layout_marginLeft**
  - **android:layout_marginRight**
- **Methods**
  - **int getLayoutDirection()**: Returns the layout direction.
  - **int getMarginEnd()**: Returns the end margin in pixels.
  - **int getMarginStart()**: Returns the start margin in pixels.
  - **void setLayoutDirection(int layoutDirection)**: Set the layout direction
  - **void setMarginEnd(int end)**: Sets the relative end margin.
  - **void setMarginStart(int start)**: Sets the relative start margin.
  - **void setMargins(int left, int top, int right, int bottom)**: Sets the margins, in pixels.
- **Fields**
  - **public int bottomMargin**: The bottom margin in pixels of the child.
  - **public int leftMargin**: The left margin in pixels of the child.
  - **public int rightMargin**: The right margin in pixels of the child.
  - **public int topMargin**: The top margin in pixels of the child.

5.7 Context

- **Methods**
  - **Resources getResources()**: Provides access to the app's resources (layouts, strings, drawables, etc.).
  - **Resources.Theme getTheme()**: : Returns the current theme for styling and inflation.

5.8 Configuration

- **Methods**
  - **public boolean isLayoutSizeAtLeast(int size)**: Checks whether the device's screen is at least a given size.
- **Fields**
  - **public int keyboard**: Keyboard for the device.
  - **public Locale locale**: User preference for the locale.
  - **public int orientation**: An integer value representing the orientation of the screen.
  - **public int screenHeightDp**: Height of the screen, not including the status bar, in dp units (density independent pixels).
  - **public int screenWidthDp**: Width of the screen in dp units.
- **Constants**
  - **ORIENTATION_PORTRAIT**: (1)
  - **ORIENTATION_LANDSCAPE**: (2)
  - **SCREENLAYOUT**$_{SIZE_{UNDEFINED}}$: (0)
  - **SCREENLAYOUT_SIZE_SMALL: (1) Smartphones**
  - **SCREENLAYOUT_SIZE_NORMAL: (2) Smartphones**
  - **SCREENLAYOUT_SIZE_LARGE: (3) Tablets**
  - **SCREENLAYOUT_SIZE_XLARGE: (4) Tablets**

5.9 Resources

- **Methods**
  - **Configuration getConfiguration()**: Return the current configuration that is
  - **DisplayMetrics getDisplayMetrics()**: Returns the current display metrics that are in effect for this resource object

5.10 Resources.Theme

- **Methods**
  - **boolean resolveAttribute(int resid, TypedValue outValue, boolean resolveRefs)**: Retrieve the value of an attribute in the Theme.

5.11 TypedValue

- **Methods**
  - **static int complexToDimensionPixelSize(int data, DisplayMetrics metrics)**: Converts a complex data value holding a dimension to its final value as an integer pixel size.
- **Constants**
  - **int COMPLEX_UNIT_SP**: TYPE_DIMENSION complex unit: Value is a scaled pixel.
  - **int COMPLEX_UNIT_PX**: TYPE_DIMENSION complex unit: Value is raw pixels.
  - **int COMPLEX_UNIT_DIP**: TYPE_DIMENSION complex unit: Value is Device Independent Pixels.

5.12 DisplayMetrics

5.13 Log

5.14 WindowManager

- **Methods**
  - **abstract Display getDefaultDisplay()**: This method was deprecated in API level 30. Use Context.getDisplay() instead.

5.15 Display

- **Methods**
  - **void getSize(Point outSize)**: This method was deprecated in API level 30. Use WindowMetrics instead. Obtain a WindowMetrics instance by calling WindowManager.getCurrentWindowMetrics(), then call WindowMetrics.getBounds() to get the dimensions of the application window.

5.16 Color

- **Methods**
  - **static int parseColor(String colorString)**: Parse the color string, and return the corresponding color-int.
- **Constants**
  - **int BLACK**:
  - **int BLUE**:
  - **int CYAN**:
  - **int DKGRAY**:
  - **int GRAY**:
  - **int GREEN**:
  - **int LTGRAY**:
  - **int MAGENTA**:
  - **int RED**:
  - **int TRANSPARENT**:
  - **int WHITE**:
  - **int YELLOW**:

5.17 ConstraintLayout

5.18 ConstraintLayout.LayoutParams

5.19 RelativeLayout

5.20 RelativeLayout.LayoutParams

5.21 LinearLayout

- **XML attributes**
  - **android:gravity**
    - center
  - **android:orientation**
    - horizontal
    - vertical
- **Methods**
  - **void setOrientation(int orientation)**: Should the layout be a column or a row.
  - **void setGravity(int gravity)**: Describes how the child views are positioned.
  - **void setHorizontalGravity(int horizontalGravity)**: Sets the horizontal gravity of the layout.
  - **void setVerticalGravity(int verticalGravity)**: Sets the vertical gravity of the layout.
  - **int getGravity()**: Returns the current gravity.
  - **int getOrientation()**: Returns the current orientation.
- **Constants**
  - **int HORIZONTAL**: Constant indicating a horizontal orientation for the layout.
  - **int VERTICAL**: Constant indicating a vertical orientation for the layout.

5.22 LinearLayout.LayoutParams

5.23 GridLayout

- **Methods**
  - **setRowCount(int rows)**
  - **setColumnCount(int cols)**
  - **void setOrientation(int orientation)**: Sets the layout's orientation. This controls:
  - **static GridLayout.Spec spec(int start, float weight)**: Equivalent to `spec(start, 1, weight)`.
  - **static GridLayout.Spec spec(int start)**: Returns a `Spec` where:
    - `span = [start, start + 1]`
    - To leave the start index undefined, use `UNDEFINED`.
  - **static GridLayout.Spec spec(int start, int size, GridLayout.Alignment alignment, float weight)**: Returns a `Spec` where:
    - `span = [start, start + size]`
    - `alignment = alignment`
    - `weight = weight`
    - To leave the start index undefined, use `UNDEFINED`.
  - **static GridLayout.Spec spec(int start, GridLayout.Alignment alignment, float weight)**: Equivalent to `spec(start, 1, alignment, weight)`.
  - **static GridLayout.Spec spec(int start, int size, GridLayout.Alignment alignment)**: Equivalent to `spec(start, size, alignment, 0f)`.
  - **static GridLayout.Spec spec(int start, GridLayout.Alignment alignment)**: Returns a `Spec` where:
    - `span = [start, start + 1]`
    - `alignment = alignment`
    - To leave the start index undefined, use `UNDEFINED`.
  - **static GridLayout.Spec spec(int start, int size, float weight)**: Equivalent to `spec(start, size, default_alignment, weight)`, where `default_alignment` is defined in `GridLayout.LayoutParams`.
  - **static GridLayout.Spec spec(int start, int size)**: Returns a `Spec` where:
    - `span = [start, start + size]`
    - To leave the start index undefined, use `UNDEFINED`.
- **Constants**:
  - **int HORIZONTAL**: Constant representing a horizontal orientation for the layout.
  - **int UNDEFINED**: Constant used to indicate that a value is undefined.
  - **int VERTICAL**: Constant representing a vertical orientation for the layout.

5.24 GridLayout.LayoutParams

5.25 GridLayout.Spec

5.26 TableLayout

5.27 TableLayout.LayoutParams

5.28 TableRow

5.29 FrameLayout

5.30 FrameLayout.LayoutParams

5.31 ListView

- **Methods**

5.32 Adapter

- **Methods**
  - **abstract int getCount():** How many items are in the data set represented by this Adapter.
  - **abstract Object getItem(int position)**: Get the data item associated with the specified position in the data set.
  - **abstract boolean isEmpty()**:
  - **abstract View getView(int position, View convertView, ViewGroup parent)**: Get a View that displays the data at the specified position in the data set.
  - **long getItemId(int position)**:

5.33 AdapterView (Abstract)

- **Methods**
  - **abstract T getAdapter()**: Returns the adapter currently associated with this widget.
  - **abstract void setAdapter(T adapter)**: Sets the adapter that provides the data and the views to represent the data in this widget.
  - **void setEmptyView(View emptyView)**: Sets the view to show if the adapter is empty
  - **View getEmptyView()**: When the current adapter is empty, the AdapterView can display a special view called the empty view.

5.34 ArrayAdapter

5.35 BaseAdapter

5.36 TextView

- **Methods**
  - **void setInputType(int type)**: Set the type of the content with a constant as defined for EditorInfo.inputType.
  - **int getInputType()**: Get the type of the editable content.
  - **public void setTextSize (float size)**:
  - **void setTextSize(int unit, float size)**: Sets text size with unit.
  - **void setGravity(int gravity)**: Sets the horizontal alignment of the text and the vertical gravity that will be used when there is extra space in the TextView beyond what is required for the text itself.
  - **void setTypeface(Typeface tf)**: Sets the typeface and style in which the text should be displayed.
  - **void setTypeface(Typeface tf, int style)**: Sets the typeface and style in which the text should be displayed, and turns on the fake bold and italic bits in the Paint if the Typeface that you provided does not have all the bits in the style that you specified.
  - **int getLineCount()**: Returns line count or 0 if layout not built.
  - **void setShadowLayer(float radius, float dx, float dy, int color)**: Applies a text shadow.

5.37 EditText

- **Methods**

5.38 InputType

- **Constants**
  - **TYPE_CLASS_TEXT**: Plain text input.
  - **TYPE_CLASS_NUMBER**: Numeric input (digits 0-9).
  - **TYPE_CLASS_PHONE**: Phone number input.
  - **TYPE_CLASS_DATETIME**: Date/time input field.
  - **TYPE_TEXT_VARIATION_NORMAL**: Default plain text.
  - **TYPE_TEXT_VARIATION_PASSWORD**: Password (hidden input).
  - **TYPE_TEXT_VARIATION_VISIBLE_PASSWORD**: Password but visible.
  - **TYPE_TEXT_VARIATION_EMAIL_ADDRESS**: Email input.
  - **TYPE_TEXT_VARIATION_PERSON_NAME**: Person's name.
  - **TYPE_TEXT_VARIATION_URI**: URL/website.
  - **TYPE_TEXT_VARIATION_POSTAL_ADDRESS**: Mailing address.
  - **TYPE_TEXT_VARIATION_SHORT_MESSAGE**: SMS/chat message.
  - **TYPE_TEXT_VARIATION_LONG_MESSAGE**: Long message or email body.
  - **TYPE_TEXT_VARIATION_WEB_EMAIL_ADDRESS**: Email in web form.
  - **TYPE_TEXT_VARIATION_WEB_PASSWORD**: Web password field.
  - **TYPE_TEXT_FLAG_CAP_SENTENCES**: Capitalize first letter of sentences.
  - **TYPE_TEXT_FLAG_CAP_WORDS**: Capitalize first letter of words.
  - **TYPE_TEXT_FLAG_CAP_CHARACTERS**: All caps.
  - **TYPE_TEXT_FLAG_MULTI_LINE**: Allows multiple lines.
  - **TYPE_TEXT_FLAG_NO_SUGGESTIONS**: Disable suggestions/auto-correct.
  - **TYPE_TEXT_FLAG_AUTO_COMPLETE**: Enable auto-complete.
  - **TYPE_TEXT_FLAG_AUTO_CORRECT**: Enable auto-correct.
  - **TYPE_NUMBER_FLAG_DECIMAL**: Allows decimal point (e.g., 3.14).
  - **TYPE_NUMBER_FLAG_SIGNED**: Allows + or – at start.
  - **TYPE_NUMBER_VARIATION_NORMAL**: Normal number input.
  - **TYPE_NUMBER_VARIATION_PASSWORD**: Number password (hidden).
  - **TYPE_DATETIME_VARIATION_NORMAL**: Default date time.
  - **TYPE_DATETIME_VARIATION_DATE**: Only date input.
  - **TYPE_DATETIME_VARIATION_TIME**: Only time input.
  - **TYPE_TEXT_FLAG_MULTI_LINE**: Allows multiple lines.
  - **TYPE_TEXT_FLAG_NO_SUGGESTIONS**: Disable suggestions/auto-correct.
  - **TYPE_TEXT_FLAG_AUTO_COMPLETE**: Enable auto-complete.
  - **TYPE_TEXT_FLAG_AUTO_CORRECT**: Enable auto-correct.
  - **TYPE_NUMBER_FLAG_DECIMAL**: Allows decimal point (e.g., 3.14).

- **TYPE_NUMBER_FLAG_SIGNED**: Allows + or – at start.
- **TYPE_NUMBER_VARIATION_NORMAL**: Normal number input.
- **TYPE_NUMBER_VARIATION_PASSWORD**: Number password (hidden).
- **TYPE_DATETIME_VARIATION_NORMAL**: Default date time.
- **TYPE_DATETIME_VARIATION_DATE**: Only date input.
- **TYPE_DATETIME_VARIATION_TIME**: Only time input.
- **TYPE_NULL**: No input type specified.
- **TYPE_MASK_CLASS**: Mask to extract base class (text/number/etc.).
- **TYPE_MASK_FLAGS**: Mask to extract all flags.
- **TYPE_MASK_VARIATION**: Mask to extract variation (like email/password).

5.39 Button

5.40 ImageView, ImageView.ScaleType

- **Methods**
  - **setImageResource(int resource)**
  - **setScaleType(ImageView.ScaleType scaleType)**
  - **public void setAdjustViewBounds (boolean adjustViewBounds)**: Set this to true if you want the ImageView to adjust its bounds to preserve the aspect ratio of its drawable.
  - **public boolean getAdjustViewBounds ()**: True when ImageView is adjusting its bounds to preserve the aspect ratio of its drawable
- **Enum values**
  - **ImageView.ScaleType CENTER**: Center the image in the view, but perform no scaling.
  - **ImageView.ScaleType CENTER_CROP**: Scale the image uniformly (maintain the image's aspect ratio) so that both dimensions (width and height) of the image will be equal to or larger than the corresponding dimension of the view (minus padding).
  - **ImageView.ScaleType CENTER_INSIDE**: Scale the image uniformly (maintain the image's aspect ratio) so that both dimensions (width and height) of the image will be equal to or less than the corresponding dimension of the view (minus padding).
  - **ImageView.ScaleType FIT_CENTER**: Scale the image using Matrix.ScaleToFit.CENTER.
  - **ImageView.ScaleType FIT_END**: Scale the image using Matrix.ScaleToFit.END.
  - **ImageView.ScaleType FIT_START**: Scale the image using Matrix.ScaleToFit.START.
  - **ImageView.ScaleType FIT_XY**: Scale the image using Matrix.ScaleToFit.FILL.
  - **ImageView.ScaleType MATRIX**: Scale using the image matrix when drawing.

5.41 ImageButton

5.42 CompoundButton

5.43 CheckBox

5.44 RadioGroup

5.45 RadioGroup.LayoutParams

5.46 RadioButton

5.47 AbsSpinner

5.48 Spinner

5.49 Progessbar

5.50 AbsSeekBar

5.51 SeekBar

5.52 Drawable

5.53 GradientDrawable

5.54 Intent

- **Methods**
  - **putExtra(String key, type value)**: Attach data to the Intent (String, int, boolean, etc.).
  - **getStringExtra(String key)**: Retrieve a String sent via Intent.
  - **getIntExtra(String key, int default)**: Retrieve an int extra.
  - **getBooleanExtra(String key, boolean default)**: Retrieve a boolean extra.
  - **getSerializableExtra(String key):** Retrieve custom objects implementing Serializable.
  - **getParcelableExtra(String key)**: Retrieve objects implementing Parcelable (faster than Serializable).
  - **getExtras()**: Get all extras as a Bundle.

5.55 Animation

5.56 AnimationSet

5.57 AlphaAnimation

5.58 RotateAnimation

5.59 ScaleAnimation

5.60 TranslateAnimation

5.61 PreferenceManager

5.62 SharedPreferences (interface)

5.63 SharedPreferences.Editor (Interface)

5.64 Menu (Interface)

5.65 MenuItem (Interface)

5.66 ContextMenu (interface)

5.67 PopupMenu

5.68 SubMenu (interface)

5.69 MenuInflator

5.70 Toast

5.71 LayoutInflator (Abstract class)

5.72 SQLiteDatabase

5.73 Cursor (Interface)

5.74 ScrollView

5.75 HorizontalScrollView

5.76 InputEvent

- **Methods**

5.77 MotionEvent

- **Methods**
  - **int getAction()**: Return the kind of action being performed.
  - **long getEventTime()**: Retrieve the time this event occurred, in the SystemClock.uptimeMillis() time base.
  - **float getRawX()**: Returns the exact place on the screen where the finger is touching.
  - **float getRawY()**: Returns the exact place on the screen where the finger is touching.
  - **float getX()**: Exact place relative to the top-left corner of the View receiving the event
  - **float getY()**: Exact place relative to the top-left corner of the View receiving the event
  - **boolean isButtonPressed(int button)**: Checks if a mouse or stylus button (or combination of buttons) is pressed.
- **Constants**
  - **int ACTION_UP**: Constant for getActionMasked(): A pressed gesture has finished, the motion contains the final release location as well as any intermediate points since the last down or move event.
  - **int ACTION_DOWN**: Constant for getActionMasked(): A pressed gesture has started, the motion contains the initial starting location.
  - **int ACTION_MOVE**: Constant for getActionMasked(): A change has happened during a press gesture (between ACTION_DOWN and ACTION_UP).
  - **int ACTION_CANCEL**: Constant for getActionMasked(): The current gesture has been aborted.
  - **int ACTION_SCROLL**: Constant for getActionMasked(): The motion event contains relative vertical and/or horizontal scroll offsets.

5.78 GestureDetector

- **Methods**
  - **boolean onTouchEvent(MotionEvent ev)**: Analyzes the given motion event and if applicable triggers the appropriate callbacks on the OnGestureListener supplied.
  - **void setOnDoubleTapListener(GestureDetector.OnDoubleTapListener onDoubleTapListener)**: Sets the listener which will be called for double-tap and related gestures.

Activity overloads

- **onCreate(Bundle savedInstanceState)**: When activity is first created (before UI is shown), Set layout, initialize variables, listeners, start essential components

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);  // Load UI
    // Initialize variables, buttons, adapters, listeners,
    ↪  etc.
}
```

- **onStart()**: Just before the activity becomes visible, Start tasks that make the activity visible (but not interactive)

```
@Override
protected void onStart() {
    super.onStart();
    // Activity is now visible (but not yet interactive)
}
```

- **onResume()**: Activity is visible and user can interact, Resume animations, sensors, camera, audio, start foreground logic

```
@Override
protected void onResume() {
    super.onResume();
    // Activity is in foreground and user can interact
    // Resume animations, sensors, cameras, music, etc.
}
```

- **onPause()**: Another activity is starting or partially covering this one, Save data, pause animations, release sensors/camera/audio

```
@Override
protected void onPause() {
    super.onPause();
    // Another activity is in front, but this one might still
    ↪  be visible
    // Pause animations, save data to SharedPreferences, stop
    ↪  camera/audio
}
```

- **onStop()**: Activity is completely hidden (not visible), Release heavy resources, stop background tasks, unregister receivers

```java
@Override
protected void onStop() {
    super.onStop();
    // Activity is no longer visible
    // Release resources that aren't needed while hidden
}
```

- **onRestart()**: When activity is coming back after being stopped, Re-initialize components if needed before onStart() is called again

```java
@Override
protected void onRestart() {
    super.onRestart();
    // Called before activity restarts after being stopped
    // Re-initialize UI elements if needed
}
```

- **onDestroy()**: Before the activity is removed from memory (finish or system kill), Final cleanup (close database, threads, etc.)

```java
@Override
protected void onDestroy() {
    super.onDestroy();
    // Final cleanup: close databases, threads, sensors
}
```

- **When do these get called**:

Situation	Lifecycle methods triggered
Open app	onCreate() → onStart() → onResume()
Press Home button	onPause() → onStop()
Return to app	onRestart() → onStart() → onResume()
Rotate screen	onPause() → onStop() → onDestroy() → onCreate()...
Open new Activity	Old: onPause() → new Activity starts → maybe onStop()
Press back (exit app)	onPause() → onStop() → onDestroy()

- **With Intents,** $A \to B$:

Event	Activity A	Activity B
A starts initially	onCreate() → onStart() → onResume()	—
$A \to B$	onPause() → onStop()	onCreate() → onStart() → onResume()
$B \to A$	onRestart() → onStart() → onResume()	onPause() → onStop() → onDestroy()

- **onConfigurationChanged**:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    ...
}
```

Listeners

- **View.OnClickListener**

```
private class ButtonHandler implements View.OnClickListener {
    public void onClick(View v) {
        ...
    }
}
```

- **RadioGroup.OnCheckedChangeListener**

```
private class RadioButtonHandler implements
↪ RadioGroup.OnCheckedChangeListener {
    public void onCheckedChanged(RadioGroup group, int
    ↪ checkedId) {
        ...
    }
}
```

- **View.OnTouchListener**: View.OnTouchListener is an interface in the Android SDK that allows your code to receive touch events occurring on a specific View.

  It defines a single callback method:

```
boolean onTouch(View v, MotionEvent event);
```

  When you implement this interface and register it on a View, Android will call your onTouch() method whenever the user touches, moves, or releases that view.

```
private class TouchHandler implements view.OnTouchListener {
    @Override
    boolean onTouch(View v, MotionEvent event) {
        ...
    }
}
```

  We return a boolean, where

  – **true**: You handled the event; stop it from going further
  – **false**: Event continues to propagate to underlying views or the view itself

  Attach the listener to a view with

```
v.setOnTouchListener(TouchHandler);
```

- **GestureDetector.OnGestureListener**: An interface that defines six callback methods used to detect general gestures such as:
  - Scroll
  - Fling
  - Long press
  - Show press
  - Single tap up
  - Down event acknowledgment

  It contains six callback methods and ALL must be implemented if you use this interface directly.

  It allows you to turn raw MotionEvent sequences into meaningful gestures such as scrolls and flings.

  The required methods are

```
boolean onDown(MotionEvent e);
void onShowPress(MotionEvent e);
boolean onSingleTapUp(MotionEvent e);
boolean onScroll(MotionEvent e1, MotionEvent e2, float
↪  distanceX, float distanceY);
void onLongPress(MotionEvent e);
boolean onFling(MotionEvent e1, MotionEvent e2, float
↪  velocityX, float velocityY);
```

  Where

  - **boolean onDown(MotionEvent e)**: Called immediately when the user first touches the screen (ACTION_DOWN).

    Initialize gesture detection, begin tracking the gesture sequence. Return true if you want to continue receiving subsequent gesture callbacks (onScroll, onFling, etc.).

    Note: This is often the first method invoked for any gesture. If you return false, some gestures may not trigger follow-up callbacks.
  - **void onShowPress(MotionEvent e)**: Called when the user has touched the screen and is holding still just long enough that the system believes a press is intentional, but before the long-press timeout is reached.

    Provide visual feedback such as highlighting a view or dimming a button. Does not confirm a click—only indicates "the user is pressing."

    Note: No return value. It does not mean the user will long-press.
  - **boolean onSingleTapUp(MotionEvent e)**: Called when the user lifts their finger after a quick tap. Simple tap recognition, UI element activation (if not using OnClickListener). Triggering lightweight actions

Note: Returning true indicates you handled the tap.

- **boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY)**: Called when the user drags/moves their finger across the screen slowly.

  Parameters:

  * **e1: the initial DOWN event**
  * **e2: the current MOVE event**
  * **distanceX: horizontal distance moved since last callback**
  * **distanceY: vertical distance moved since last callback**

  Note: This method fires repeatedly during movement. It represents a scroll gesture, not a fling.

  e2 is the latest MotionEvent delivered to GestureDetector while the user's finger is moving across the screen. Every time the gesture detector detects movement, it generates a new call to onScroll, and e2 is the MotionEvent from that most recent ACTION_MOVE.

  This continues until the user lifts their finger (ACTION_UP), at which point scrolling stops.

- **void onLongPress(MotionEvent e)**: Called when the user has pressed and held for the long-press timeout, typically ∼500 ms.

  This is the exact moment Android recognizes a long-press gesture.

- **boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)**: Called when the user quickly swipes and lifts their finger with momentum (fast movement + release).

  Parameters:

  * **e1: initial DOWN event**
  * **e2: final UP event**
  * **velocityX: horizontal swipe velocity (pixels/second)**
  * **velocityY: vertical swipe velocity (pixels/second)**

  A fling is not just a scroll; it requires sufficient speed.

- **GestureDetector.OnDoubleTapListener**: GestureDetector.OnDoubleTapListener is an interface that allows you to receive callbacks specifically related to double-tap gestures and related tap-state events. It is an optional companion to a GestureDetector.OnGestureListener. You attach it to a GestureDetector when you want to detect

  - A confirmed double tap
  - The separate events that happen before and after a double tap
  - The two distinct tap actions inside that double tap

  This interface is only used when you care about double-tap-based interaction, such as zooming, selecting, or triggering special UI actions.

  The three methods are

```
boolean onSingleTapConfirmed(MotionEvent e);
boolean onDoubleTap(MotionEvent e);
boolean onDoubleTapEvent(MotionEvent e);
```

Where

- **boolean onSingleTapConfirmed(MotionEvent e)**: This is triggered when the system is sure the gesture was a single tap, and not part of a double tap. It fires after the double-tap timeout passes.

  Use this for actions where you need to distinguish between single tap and double tap. For example, a single tap selects an item, but a double tap opens it. You must not perform the single-tap action immediately; you must wait to see if a second tap arrives.
- **boolean onDoubleTap(MotionEvent e)**: This triggers when the user taps twice in quick succession according to Android's double-tap detection timing. This method fires once for the entire double-tap gesture (i.e., when the second tap occurs). If you want to zoom in/out or run any "double-tap special action," this is the correct place.
- **boolean onDoubleTapEvent(MotionEvent e)**: This method receives all events that occur during the double-tap gesture, including:
  * ACTION_DOWN of the second tap
  * ACTION_MOVE events between the two taps
  * ACTION_UP events
  * Small drags during a double-tap sequence

  If you only need simple double-tap detection, you do not implement this method.

You attach the listener to a GestureDetector instance

```
GestureDetector detector = new GestureDetector(this,
↪  gestureListener);
detector.setOnDoubleTapListener(doubleTapListener);
```

Where `gestureListener` is a class that implements GestureDetector.OnGestureListener, and `doubleTapListener` is a class that implements GestureDetector.OnDoubleTapListener.

The GestureDetector then routes appropriate touch events to it when you forward all motion events from your View or Activity:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    return detector.onTouchEvent(event);
}
```

By doing this, all touch events inside the activity will be handled by the GestureDetector and its provided listeners. If we wanted to handle touch events only when they happen inside a specific view, we can do

```
myView.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent e) {
        detector.onTouchEvent(e);
        return true;
    }
});
```

Created Transitions

Notes

9.1 View

- **Measuring views**: measure() is part of the layout pass in Android. It tells a View how big it wants to be given the constraints provided by its parent.

  When you call measure(widthSpec, heightSpec) yourself, you are manually triggering its measurement phase.

  A measurement includes:

  – the parent's constraints (the measure specs),

  – the view's own content and layout parameters,

  – and any minimum/maximum size rules.

  measure() does not set the final size for drawing. It only calculates the desired size. The final size is set later by layout().

  After measure() runs, the view stores the results internally. We retrieve them using

```
int w = view.getMeasuredWidth();
int h = view.getMeasuredHeight();
```

  These values represent the size the view calculated for itself based on the constraints given in measure().

  They are not the final displayed size on screen, they are the measured size that the view wants.

  The true on-screen size is obtained from:

```
view.getWidth();
view.getHeight();
```

  Consider the example

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/
↪ res/android"
    android:id="@+id/root"
    android:layout_width="200dp"     <!-- Parent is ONLY
    ↪ 200dp wide -->
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <Button
        android:id="@+id/myButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="THIS IS A VERY LONG BUTTON TEXT" />

</LinearLayout>
```

A Button inside a narrow LinearLayout tries to become very wide (because of its text), but the parent forces it to fit into a smaller width.

The Button measures itself by looking at the text. It may report a measured width of, say, 600px (because the text is long), but the parent is only 200dp wide, and it must fit inside it. So, the parent forces the Button to 200dp – padding – margin.

Note: We can use this information about measuring views to get the size of a view that has yet to be displayed.

- **boolean onTouchEvent(MotionEvent event)**: onTouchEvent is a built-in method of every View in Android.

  Its purpose is to:

  1. Receive low-level touch events (DOWN, MOVE, UP) that reach the view.
  2. Decide whether the view will handle them or ignore them.
  3. Trigger higher-level behaviors such as clicks, long-presses, scrolling, focusing, etc.

  Every custom view and every standard widget (Button, TextView, ImageView, etc.) has its own implementation of this method.

  The sequence is:

  1. User touches screen → system creates a MotionEvent.
  2. Android routes the event to the view under the touch.
  3. If a View.OnTouchListener is attached:
     - Android calls listener.onTouch(v, event) first.
     - If that method returns true, the event is consumed, and view.onTouchEvent() will NOT be called.
  4. If the listener returns false, then Android calls the view's own onTouchEvent(event).

  The default implementation in View handles

  1. **Click detection**: If the sequence of MotionEvents is DOWN, optional small MOVE, UP, then onTouchEvent interprets that as a click and calls

```
performClick();
```

which triggers

```
OnClickListener.onClick(view)
```

If you consume the event in a touch listener (returning true), this will not run.

2. **Long-press detection**: If the finger stays down long enough, it triggers:

```
performLongClick();
```

Which triggers any OnLongClickListener.

3. **Focusable and selectable states**
4. **Scrolling for scrollable views**

If you are creating a custom view, you typically override onTouchEvent when you need to take full control of touch behavior.

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
        // Start tracking
        return true;   // Important: we claim the gesture

        case MotionEvent.ACTION_MOVE:
        // Update UI or internal state
        break;

        case MotionEvent.ACTION_UP:
        // Perform an action if needed
        performClick();
        break;
    }

    return super.onTouchEvent(event);
}
```

We return true on ACTION_DOWN because it tells android "This view is handling this gesture; send me all future MOVE and UP events." If you return false on ACTION_DOWN, Android stops sending you the rest of the gesture.

- **Why onTouchEvent Matters When Using GestureDetector**: When we do

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    return gestureDetector.onTouchEvent(event);
}
```

You are effectively saying:

- "GestureDetector will decide what to do with this event."
- "GestureDetector's callbacks (onScroll, onFling, etc.) will replace default View behavior."

9.2 LinearLayout

- **Specifying where all children go**: The `LinearLayout` has an XML attribute

  ```
  o  android:gravity
  ```

  With possible values

Constant	Value	Description
bottom	50	Push object to the bottom of its container, not changing its size.
center	11	Place the object in the center of its container in both the vertical and horizontal axis, not changing its size.
center_horizontal	1	Place object in the horizontal center of its container, not changing its size.
center_vertical	10	Place object in the vertical center of its container, not changing its size.
clip_horizontal	8	Additional option that can be set to have the left and/or right edges of the child clipped to its container's bounds. The clip will be based on the horizontal gravity: a left gravity will clip the right edge, a right gravity will clip the left edge, and neither will clip both edges.
clip_vertical	80	Additional option that can be set to have the top and/or bottom edges of the child clipped to its container's bounds. The clip will be based on the vertical gravity: a top gravity will clip the bottom edge, a bottom gravity will clip the top edge, and neither will clip both edges.
end	800005	Push object to the end of its container, not changing its size.
fill	77	Grow the horizontal and vertical size of the object if needed so it completely fills its container.
fill_horizontal	7	Grow the horizontal size of the object if needed so it completely fills its container.
fill_vertical	70	Grow the vertical size of the object if needed so it completely fills its container.
left	3	Push object to the left of its container, not changing its size.
right	5	Push object to the right of its container, not changing its size.
start	800003	Push object to the beginning of its container, not changing its size.
top	30	Push object to the top of its container, not changing its size.

  The `android:gravity` attribute in a `LinearLayout` controls how the layout positions its own children inside itself. This is different from layout_gravity, which controls how the view itself is positioned inside its parent.

- **Specifying where a specific child view goes**: For this, we use the

  ```
  o  android:layout_gravity
  ```

  attribute on the child view. This attribute is part of `LinearLayout.LayoutParams`

9.3 GridLayout

- **Setting orientation**: n GridLayout (Java), you set the orientation directly on the GridLayout instance. The orientation determines how automatic placement proceeds—either left-to-right then top-to-bottom (horizontal), or top-to-bottom then left-to-right (vertical).

  The two valid constants are:

  - GridLayout.HORIZONTAL
  - GridLayout.VERTICAL

- **Positioning**: In GridLayout, every child view lives in a "cell" identified by a (row, column) pair, with indices starting at 0. Positioning is controlled by GridLayout.LayoutParams, specifically its rowSpec and columnSpec fields.

  There are two ways in which we can position items in a GridLayout

  1. **Default**: Let GridLayout place in the next available position. The Grid
  2. **Specified**: We can specify an exact (row, col) position to place views at.

- **The GridLayout.Spec object**: A GridLayout.Spec is an object that describes how a single child occupies the grid—specifically, which rows or columns it occupies, how many cells it spans, and what its alignment is within those cells.

  A child has two specs:

  1. **rowSpec**: Defines row index and row span
  2. **columnSpec**: Defines column index and column span

  These two together fully describe the child's position in the grid. We give these two Spec objects to the GridLayout.LayoutParams that we attach to a child view.

  A Spec object encodes three things

  1. **Start index**: The row or column where the child begins
  2. **Span**: How many rows or columns it covers
  3. **Alignment (optional)**: How the child aligns inside its cells

  We create a GridLayout.Spec object by calling GridLayout.spec(...)

- **Default positioning**:

```
0  // rowSpec and columnSpec say where this view goes
1  GridLayout.Spec rowSpec =
   ↪  GridLayout.spec(GridLayout.UNDEFINED);
2  GridLayout.Spec colSpec =
   ↪  GridLayout.spec(GridLayout.UNDEFINED);
3
4  GridLayout.LayoutParams params = new
   ↪  GridLayout.LayoutParams(rowSpec, colSpec);
5
6  grid.addView(v, params);
```

`GridLayout.UNDEFINED` Means "I am not specifying this index; let GridLayout figure it out."

- **Automatic placement with spanning**: Now suppose you want a view that spans multiple cells but you still do not care where it lands, only that a block of that size fits.

  Example: automatically place a view that is 2 columns wide and 1 row high.

```
// row index: unspecified, 1 row high
GridLayout.Spec rowSpec =
    GridLayout.spec(GridLayout.UNDEFINED, 1);

// column index: unspecified, span 2 columns
GridLayout.Spec colSpec =
    GridLayout.spec(GridLayout.UNDEFINED, 2);
```

- **Explicit positioning (specific row and column)**: To place a view in a specific cell (row, column):

```
// Place at row 1, column 2 (second row, third column)
GridLayout.Spec rowSpec = GridLayout.spec(1); // row index 1
GridLayout.Spec colSpec = GridLayout.spec(2); // column index
    2

GridLayout.LayoutParams params = new
    GridLayout.LayoutParams(rowSpec, colSpec);
grid.addView(button, params);
```

- **(android.widget.GridLayout) Getting the position of a view in a GridLayout**: The API does not expose publicly which position a view is placed in, so we must track this information ourselves

- **(androidx.gridlayout.widget.GridLayout) Getting the position of a view in a GridLayout**: You obtain the (row, column) pair of a child view by reading its GridLayout.LayoutParams and then inspecting the rowSpec and columnSpec inside those layout parameters.

  Because GridLayout stores position metadata in the Spec objects, this is the only correct way to retrieve a view's location.

```
View child = ...;                  // the view inside the
    GridLayout
GridLayout.LayoutParams lp = (GridLayout.LayoutParams)
    child.getLayoutParams();

GridLayout.Spec rowSpec = lp.rowSpec;
GridLayout.Spec colSpec = lp.columnSpec;
```

Each Spec contains:

- start index
- span length
- alignment information

Unfortunately, the start and span fields are not public fields. But Android provides getter methods indirectly via Spec:

- **Starting row / column indices**:

```
int rowSpan = rowSpec.getSpan().size();
int colSpan = colSpec.getSpan().size();
```

- **Ending row / column indices**:

```
int rowEnd = rowSpec.getSpan().max;   // inclusive
int colEnd = colSpec.getSpan().max;   // inclusive
```

- **Span length**:

```
int rowSpan = rowSpec.getSpan().size();
int colSpan = colSpec.getSpan().size();
```

- **Alignment**:

```
GridLayout.Alignment align = rowSpan.getAlignment();
```

- **Getting child at (row, column) position**: here is no built-in API that maps (row, column) → child view, we must track this information ourselves.