Lecture 2: Layouts, Drawable and Guidelines

- **Mobile Application Development (COMP2008)**

- **Discipline of Computing School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)**

(Windows 1.0.1 demo slideshow.)

Outline

Layouts

Drawable

Different Layouts

Guideline

User Interfaces

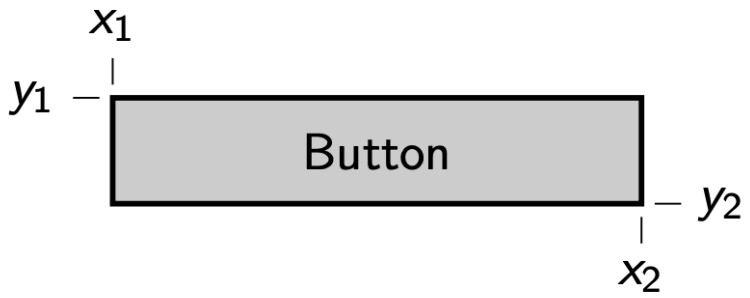
- UIs are deceptively complex.
  - (Seeing all the options in Android Studio hints at this complexity.)
- There are many types of UI elements.
  - Each has many properties.
- But the complexity is mostly due to layouts – where things go.

Containers and Child Elements

- GUIs (including mobile UIs) are hierarchical – they're trees1 .
- They may look just like a rectangle with stuff in it. . .
- . . . But there are groups of UI elements, groups within groups, and so on.
- In Android:
  - Button, EditText, ViewText, etc. are kinds of "View".
  - "ViewGroup" is also a kind of View that contains other Views.
- I In iOS:
  - UIButton, UITextField, etc. are kinds of "UIView"s.
  - Any UIView can contain other UIViews (but only some actually do, in practice).
- All UI elements occupy a rectangular area of the screen.
- Container elements are rectangles that "fence in" their child elements.

Layouts

- All UI elements are rectangles, so it takes 4 numbers to represent their size and position:
- But UI elements must also adapt to different screen sizes!
- Different devices, rotated devices, split screen, etc.
- x1, y1, x2 and y2 cannot be fixed at compile time.
- They must be calculated when the GUI is displayed.
- Container elements “decide” how their child elements are sized and positioned within them.

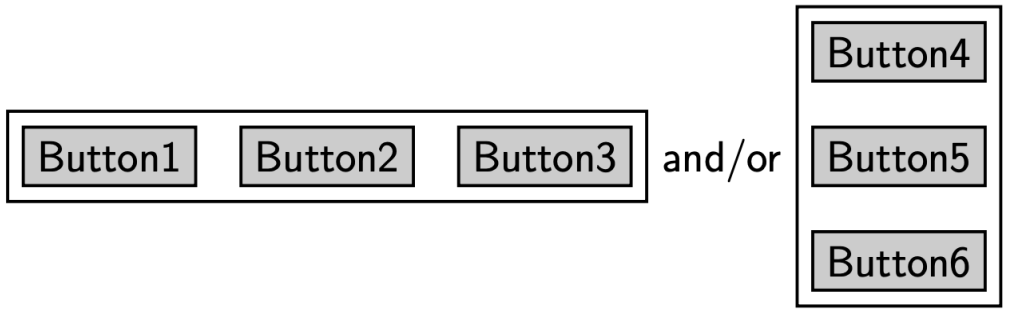


Layouts

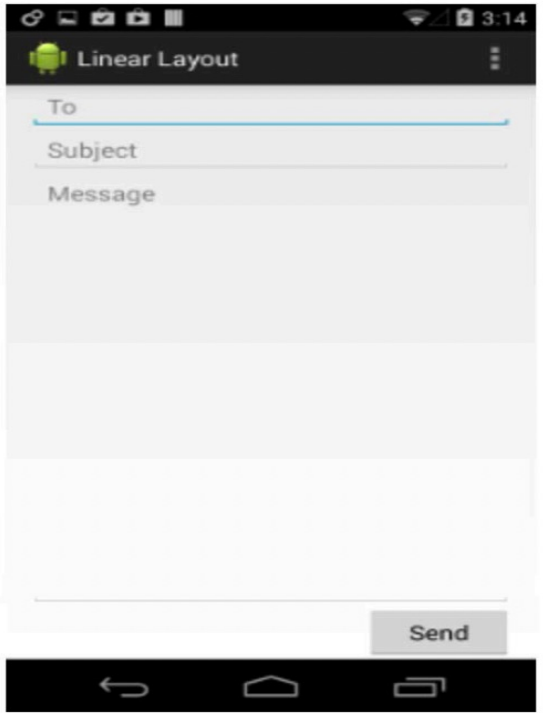
- Android Studio has a “drag and drop” UI editor.
  - Can be deceptive! You cannot simply place a view at a particular (x, y) location.
  - This would break on different screen sizes.
  - Instead, you put views into view groups, and the view groups dynamically figure out where everything goes.
- In Android:
  - Different subclasses of ViewGroup lay out things differently.
  - LinearLayout is a container that arranges its elements in a line, horizontally or vertically.
  - ConstraintLayout is a container that places elements relative to each other, based on a set of “constraints” that you specify.
- In iOS:
  - UIStackView is comparable to Android’s LinearLayout.
  - The “Auto Layout” feature achieves something like ConstraintLayout, but isn’t directly tied to the view hierarchy

LinearLayout (Android) and UIStackView (iOS)

- Horizontal/vertical stacking of UI elements – simple and powerful.
- They can be nested – you can put a horizontal panel inside a vertical one, and vice versa.
- Size and “padding” can be adjusted to fit the available space.



```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```



Source: https://developer.android.com/guide/topics/ui/layout/linear

What is DP?

"dp" stands for Density-independent Pixels. It is a unit of measurement used to express distances, dimensions, and sizes in a way that allows the user interface to scale appropriately across different screen densities.

Screen density refers to the number of pixels within a given area on the screen. Devices with higher densities have more pixels per inch, and devices with lower densities have fewer pixels per inch.

If you define a view's size in dp, it will appear roughly the same physical size on different devices, regardless of their screen density.

To convert dp to pixels (px), Android uses a scaling factor based on the device's screen density. The formula to convert dp to px is:

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

android:layout_width and android:layout_height

These attributes are essential for positioning and sizing views correctly within a user interface.

- match_parent: This value makes the view fill its parent's width (for android:layout_width) or height (for android:layout_height). The view expands to take all available space in the specified dimension.
- wrap_content: This value makes the view size itself to fit its content. The view will be as wide or tall as needed to accommodate its content.
- specific_size: You can also specify an exact size for the view in pixels or density-independent pixels (dp). For example, android:layout_width="200dp" sets the width of the view to 200 dp.
- 0dp or 0px: Setting the width or height to 0dp (or 0px) allows you to use constraints in ConstraintLayout. The view will be sized based on the constraints you define. allowing the view to expand or shrink based on the available space and the constraints you set for it

Linear Layout options

The android:layout_weight attribute takes a floating-point value (typically greater than 0) that represents the proportion of the available space a particular view should occupy relative to other views in the same parent container.

If you set android:layout_width or android:layout_height of the child views to 0dp (or match_parent along with android:layout_weight="1"), the system will distribute the available space based on the weights specified.

android:gravity is used to control the alignment of the View's content inside the View itself.

android:layout_gravity is used to control the alignment of the View within its parent container.

Try Yourself - Draw the layouts

1

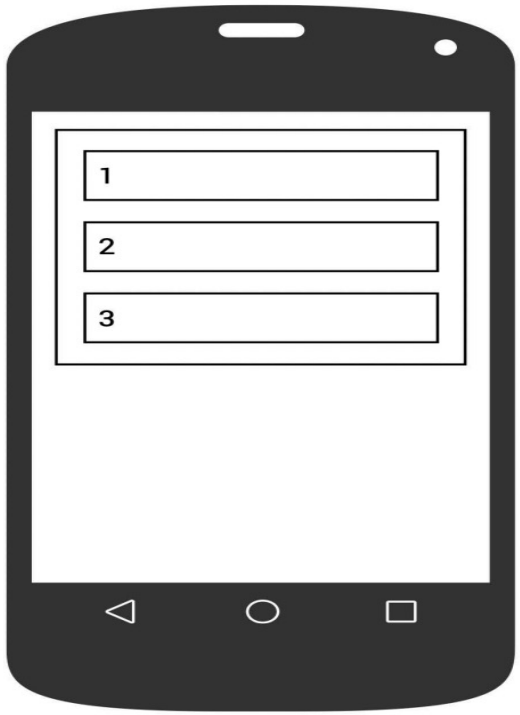
```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="1"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="2"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="3"/>

</LinearLayout>
```

2

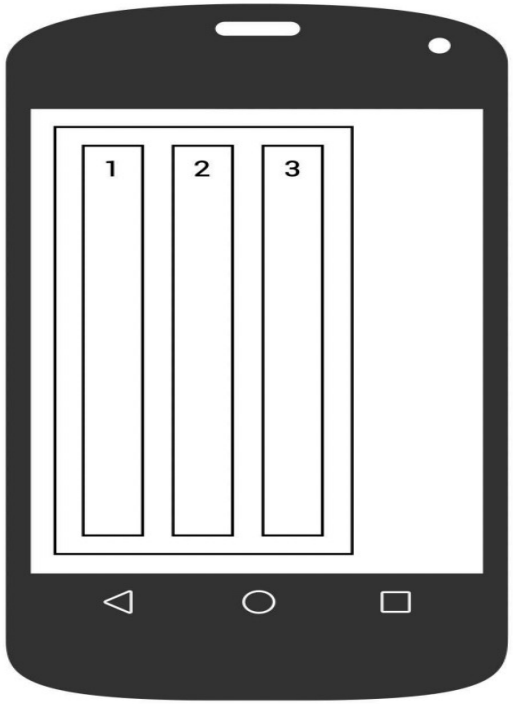
```
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:text="1"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:text="2"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:text="3"/>

</LinearLayout>
```

Source: https://developer.android.com/guide/topics/ui/layout/linear

1

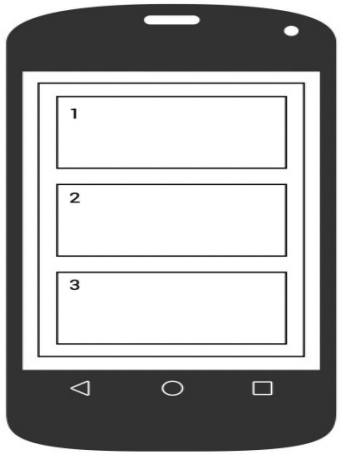


2

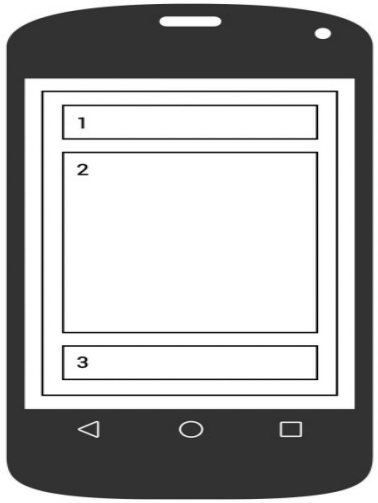


Source: https://developer.android.com/guide/topics/ui/layout/linear

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="1"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="2"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="3"/>

</LinearLayout>
```




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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:text="1"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:text="2"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:text="3"/>

</LinearLayout>
```

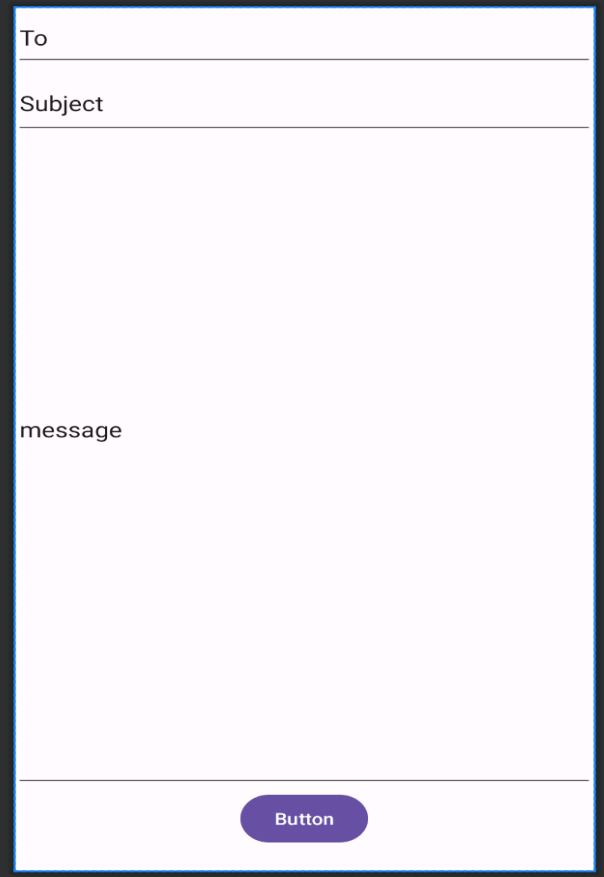


ConstraintLayout (Android) and Auto Layout (iOS)

- More complex, but more flexible.
- Each view inside a ConstraintLayout has one or more constraints.
  - These define its position, relative to something else.
- We place each UI element relative to something else; e.g.
  - At the top-left of the container;
  - In the centre of the container;
  - To the right of another element (that has already been positioned).
- The drag-and-drop editor shows these constraints visually, and lets you create/delete them.

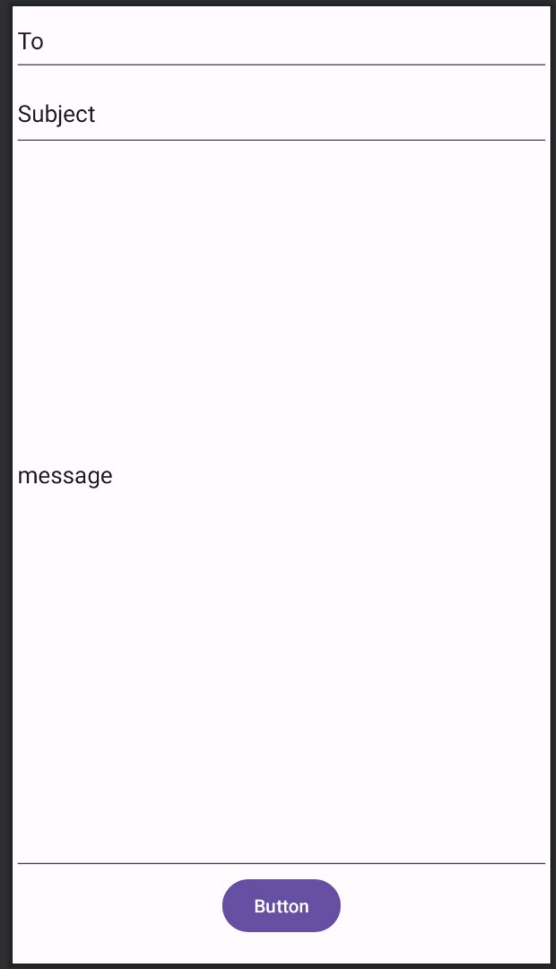
Example

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:a
    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">

    <EditText
        android:id="@+id/to"
        android:layout_width="0dp"
        android:layout_height="53dp"
        android:ems="10"
        android:inputType="text"
        android:text="To"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/subject"
        android:layout_width="0dp"
        android:layout_height="58dp"
        android:ems="10"
        android:inputType="text"
        android:text="Subject"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/to" />
```



Example
(Cont..)

```
    <EditText
        android:id="@+id/message"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:ems="10"
        android:inputType="text"
        android:text="message"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/to"
        app:layout_constraintBottom_toTopOf="@+id/button"/>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="20dp"
        android:text="Button"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />


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



Alternate UI Layouts

- Flexible layouts (linear, constraint-based, or others) help make adaptable UIs.
- Basically, a single layout can be stretched or squashed to fit different screens.
- But there are limits to this.
  - A big/complex layout may simply not fit on a small screen.
  - A small/simple layout may waste space on a large screen.
- It's possible (and often good!) to have alternative layouts; e.g.:
  - One for watches, one for phones, one for tablets, one for TVs; or
  - One for portrait, one for landscape.

Alternate UI Layouts: Android

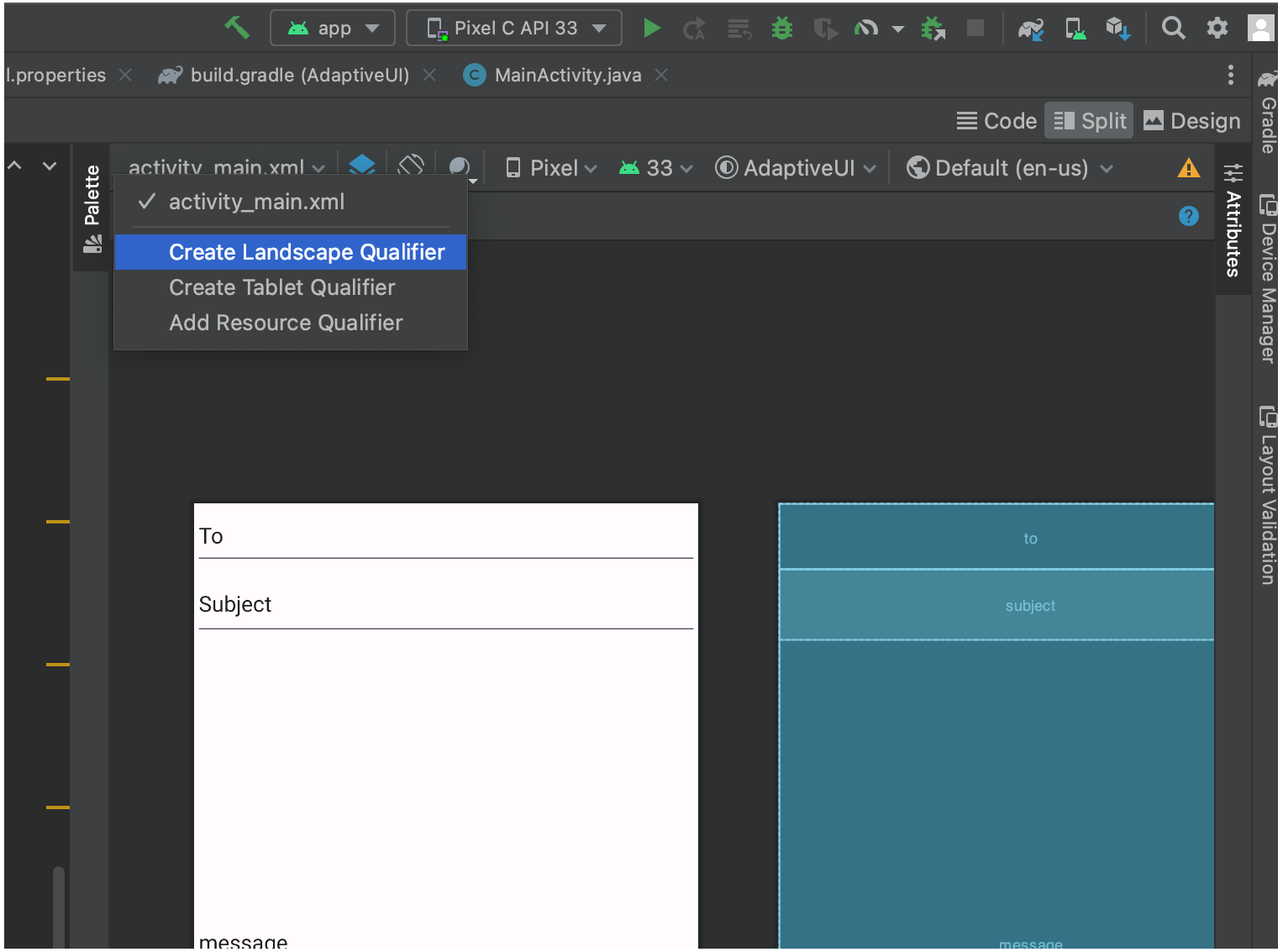
- Android automatically selects between different layouts as follows:
  - app/src/main/res/layout/ contains the “default” UI layout XML file(s).
  - An alternate set of XML files can exist in app/src/main/res/layout-qualifier/.
  - Where “qualifier” could be, for instance:
    - large for tablet-sized devices or larger;
    - sw600dp for screens whose width and height are each at least 600 “dp” units;
    - land for landscape orientation;
    - notouch for non-touch screens; . . .
    - and many others2 .
- Combos are possible; e.g. .../res/layout-sw450dp-land.

---

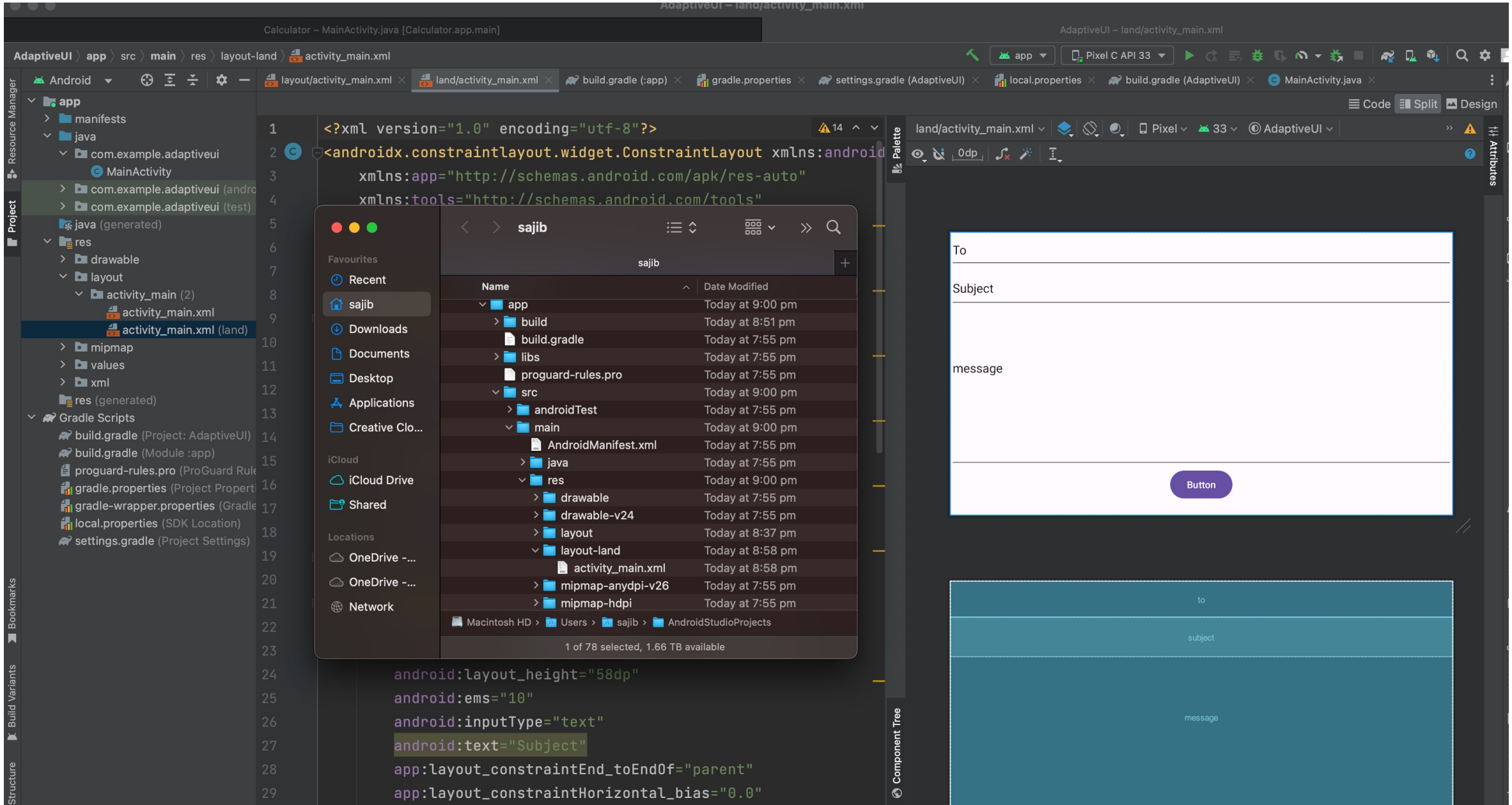
1 https://developer.android.com/training/multiscreen/screensizes

2 https://developer.android.com/guide/topics/resources/providing-resources 1

Creating landscape layout



Landscape example

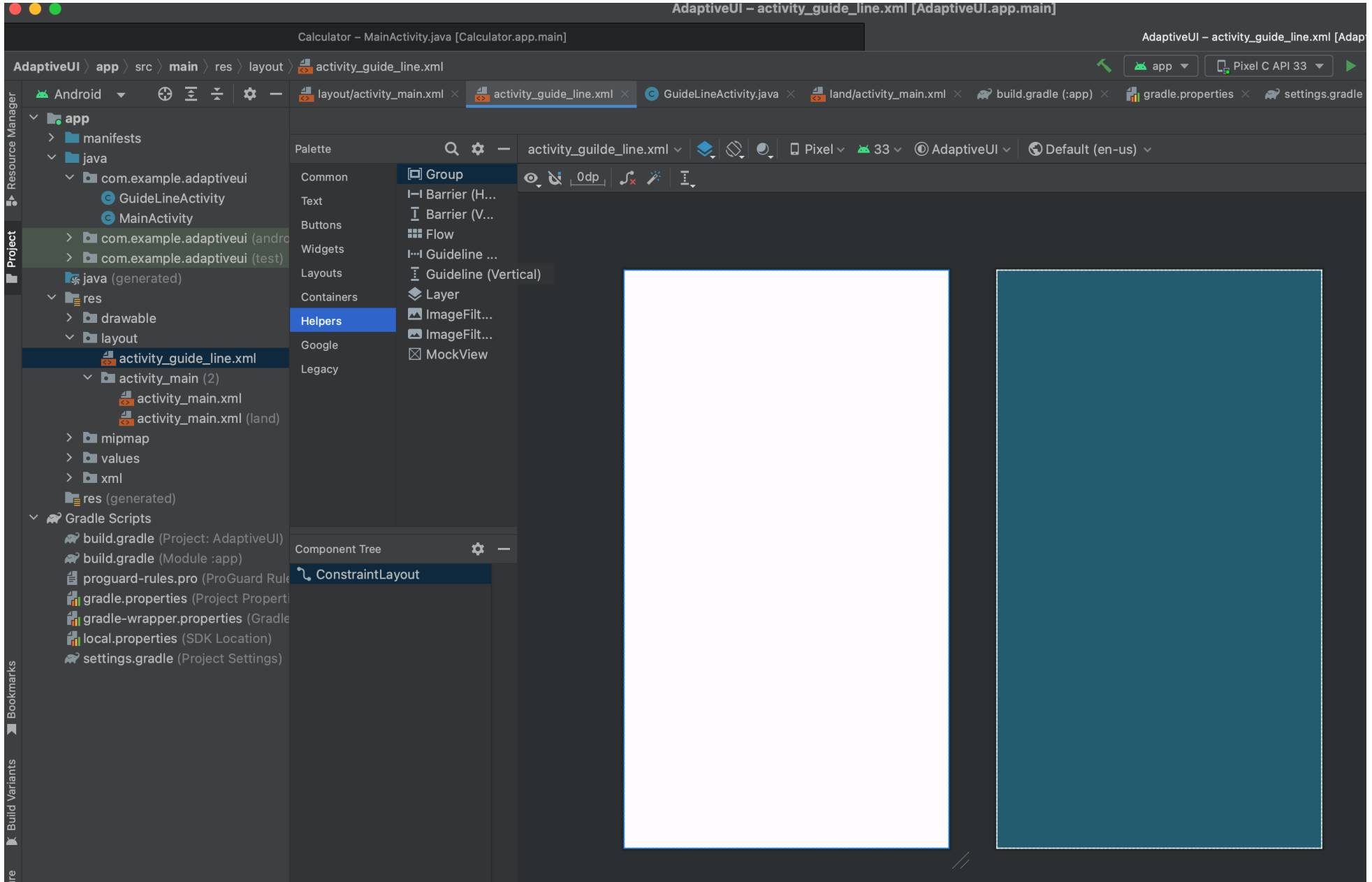


androidx.constraintlayout.widget.Guideline

It allows you to create a reference line (a guideline) within a ConstraintLayout to help position and align other views more easily. Guideline is an invisible view, meaning it doesn't appear on the screen; its purpose is solely for layout purposes.

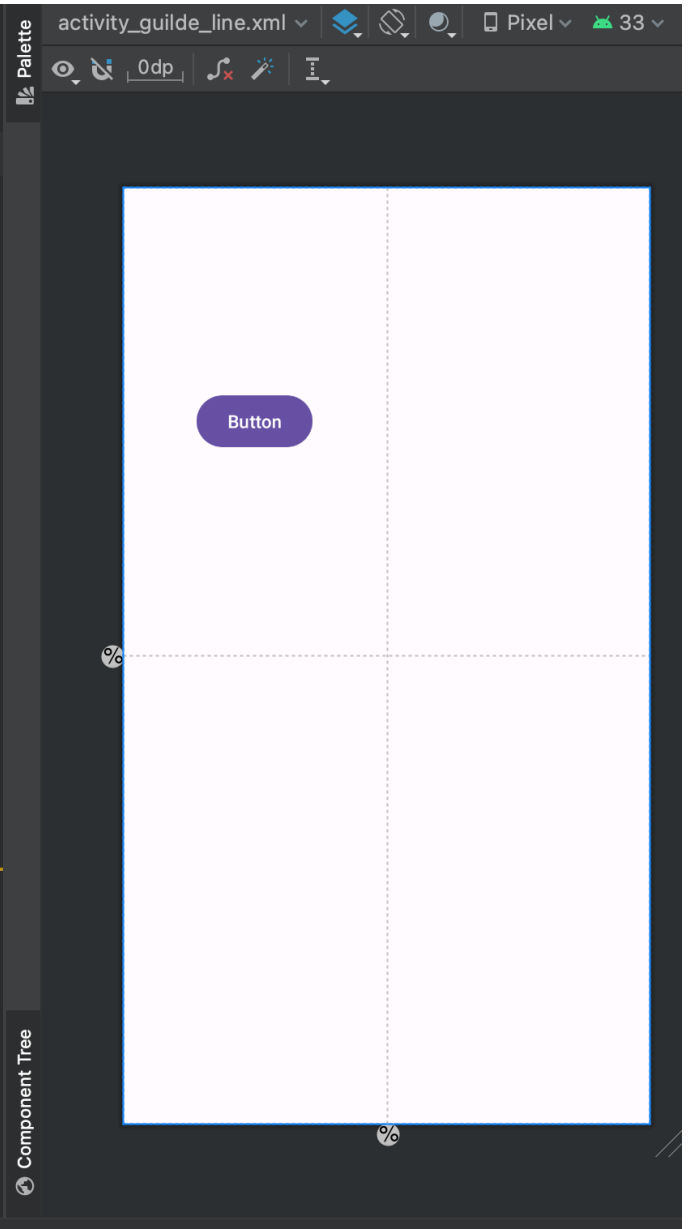
app:layout_constraintGuide_percent attribute, which accepts a float value between 0 and 1. The value represents the percentage of the ConstraintLayout's width (for a vertical guideline) or height (for a horizontal guideline) where the guideline should be placed.

Guidelines


AdaptiveUI – activity_guide_line.xml [AdaptiveUI.app.main]
Calculator – MainActivity.java [Calculator.app.main]
AdaptiveUI – activity_guide_line.xml [Adap
AdaptiveUI 〉 app 〉 src 〉 main 〉 res 〉 layout 〉 activity_guide_line.xml
app
Pixel C API 33
Android
layout/activity_main.xml
activity_guide_line.xml
GuideLineActivity.java
land/activity_main.xml
build.gradle (:app)
gradle.properties
settings.gradle
Resource Manager
Project
app
manifests
java
com.example.adaptiveui
GuideLineActivity
MainActivity
com.example.adaptiveui (andro
com.example.adaptiveui (test)
java (generated)
res
drawable
layout
activity_guide_line.xml
activity_main (2)
activity_main.xml
activity_main.xml (land)
mipmap
values
xml
res (generated)
Gradle Scripts
build.gradle (Project: AdaptiveUI)
build.gradle (Module :app)
proguard-rules.pro (ProGuard Rule
gradle.properties (Project Properti
gradle-wrapper.properties (Gradle
local.properties (SDK Location)
settings.gradle (Project Settings)
Palette
Common
Text
Buttons
Widgets
Layouts
Containers
Helpers
Google
Legacy
Group
Barrier (H...
Barrier (V...
Flow
Guideline ...
Guideline (Vertical)
Layer
ImageFilt...
ImageFilt...
MockView
Component Tree
ConstraintLayout
activity_guilde_line.xml
Pixel
33
AdaptiveUI
Default (en-us)
0dp
Bookmarks
Build Variants

Guidelines (example)

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".GuideLineActivity">

    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        app:layout_constraintGuide_percent=".5" />

    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/guideline2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintGuide_percent=".5"/>

    <Button
        android:id="@+id/button2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        app:layout_constraintBottom_toTopOf="@+id/guideline2"
        app:layout_constraintEnd_toStartOf="@+id/guideline1"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```



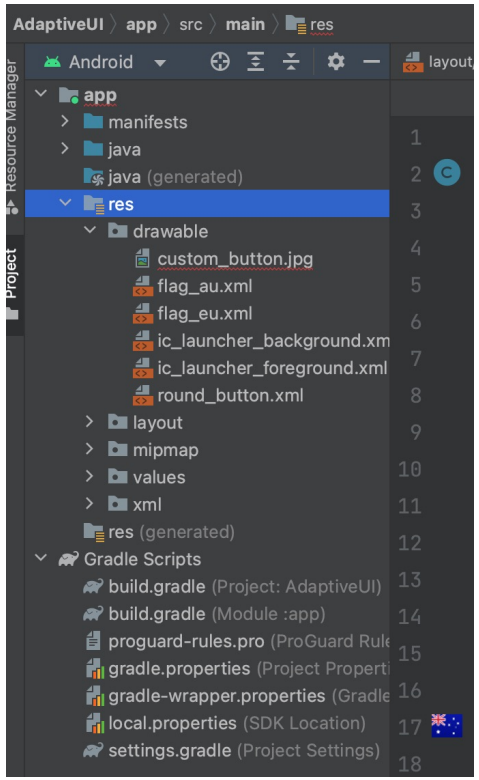
Drawables

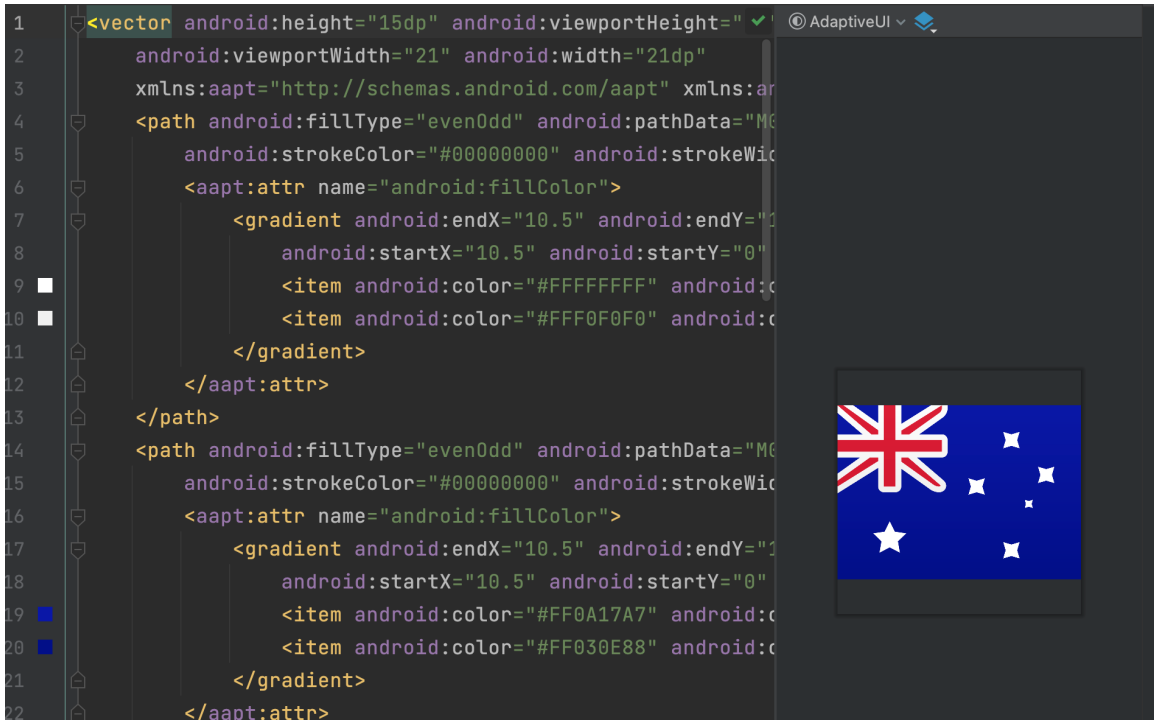
Drawables are graphic resources used to define visual elements, such as images, shapes, gradients, and colors, for various UI components and backgrounds.

- Bitmap Drawables: These are raster images (PNG, JPEG, etc.) that can be used for icons, images, and other graphics. Bitmap drawables are typically stored in the res/drawable directory.
- Vector Drawables: Vector drawables are XML-based graphics that use scalable vector graphics (SVG) format. They are resolution-independent and can be scaled without losing quality. Vector drawables are stored in the res/drawable directory as well and have the file extension .xml.
- Shape Drawables: Shape drawables are XML-based graphics that allow you to define simple shapes, such as rectangles, ovals, lines, and gradients. They are useful for creating backgrounds or simple icons. Shape drawables are stored in the res/drawable directory as XML files.

Adding drawables

- Just copy the JPG/PNG or vector drawables (.xml files) directly to /res/drawbale



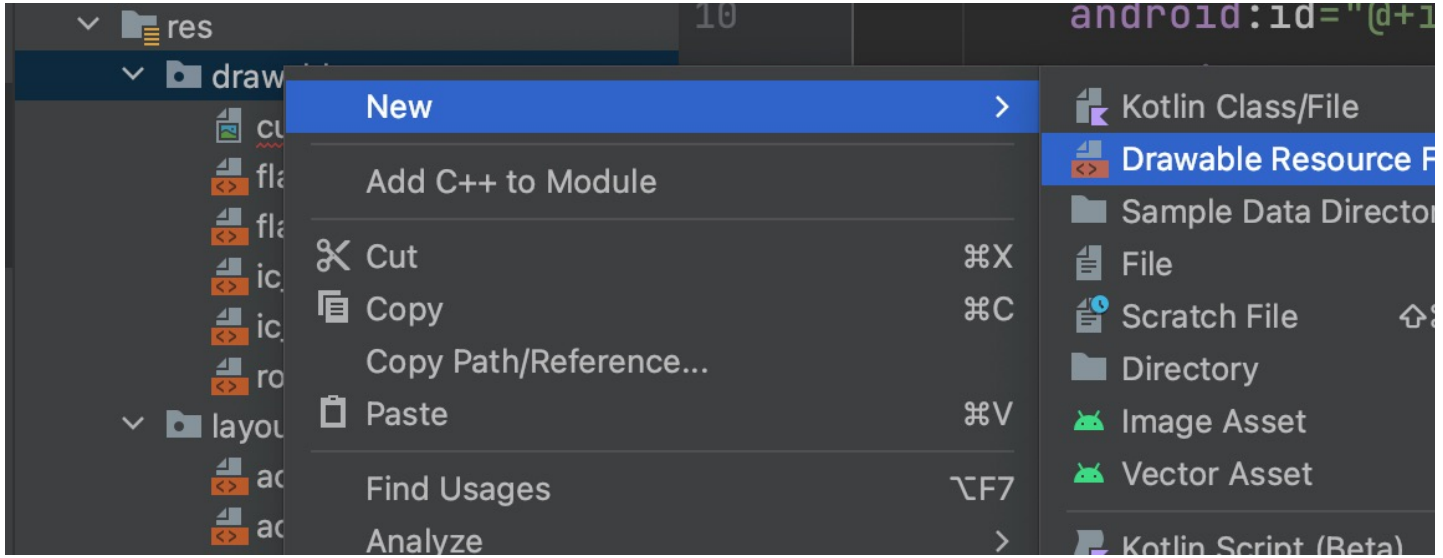


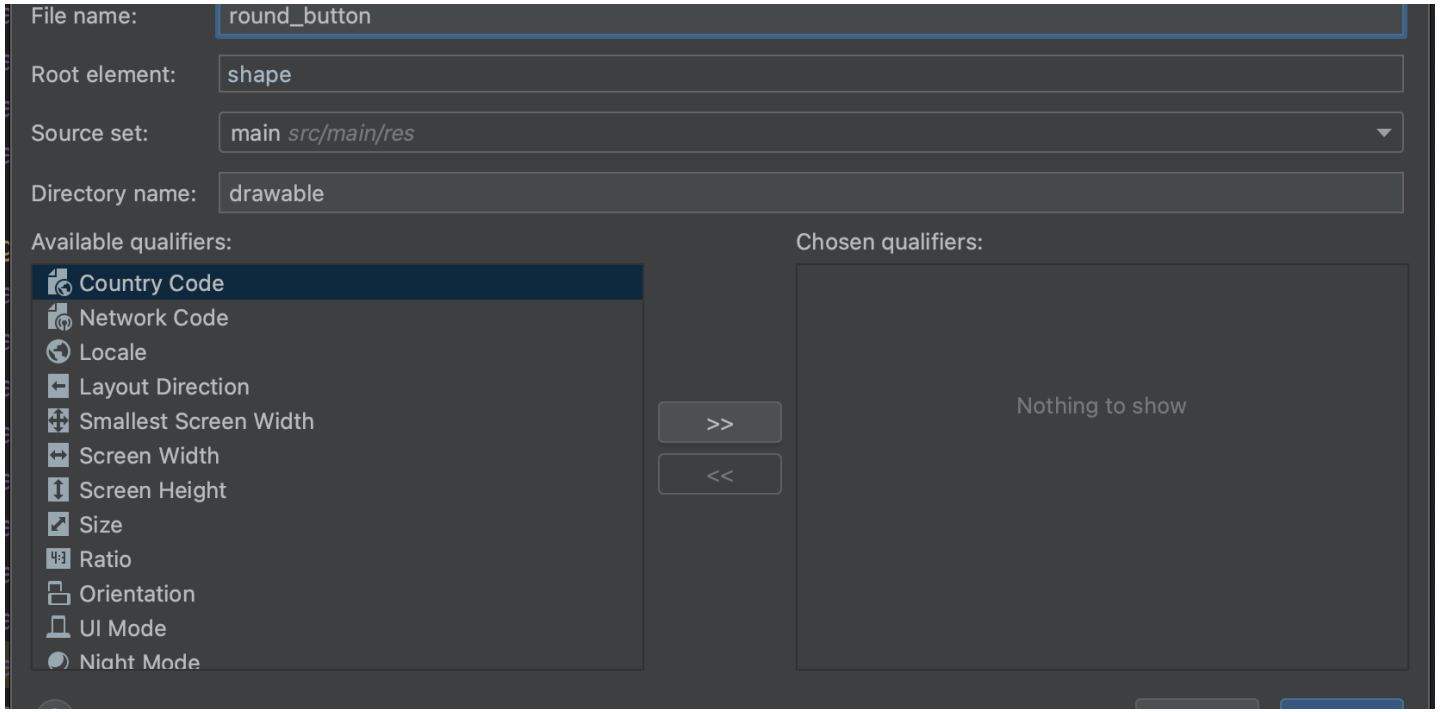
Working with ImageView and ImageButton

```
<ImageView
    android:id="@+id/imageView5"
    android:layout_width="171dp"
    android:layout_height="160dp"
    android:layout_marginTop="30dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/flag_au" />

<ImageButton
    android:id="@+id/imageButton2"
    android:layout_width="201dp"
    android:layout_height="202dp"
    android:layout_marginTop="30dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/imageView5"
    app:srcCompat="@drawable/custom_button"
    android:contentDescription="Home"
    android:scaleType = "fitXY"/>
```

Working with Shapes

- You can create your shapes and use them as a background to other views.

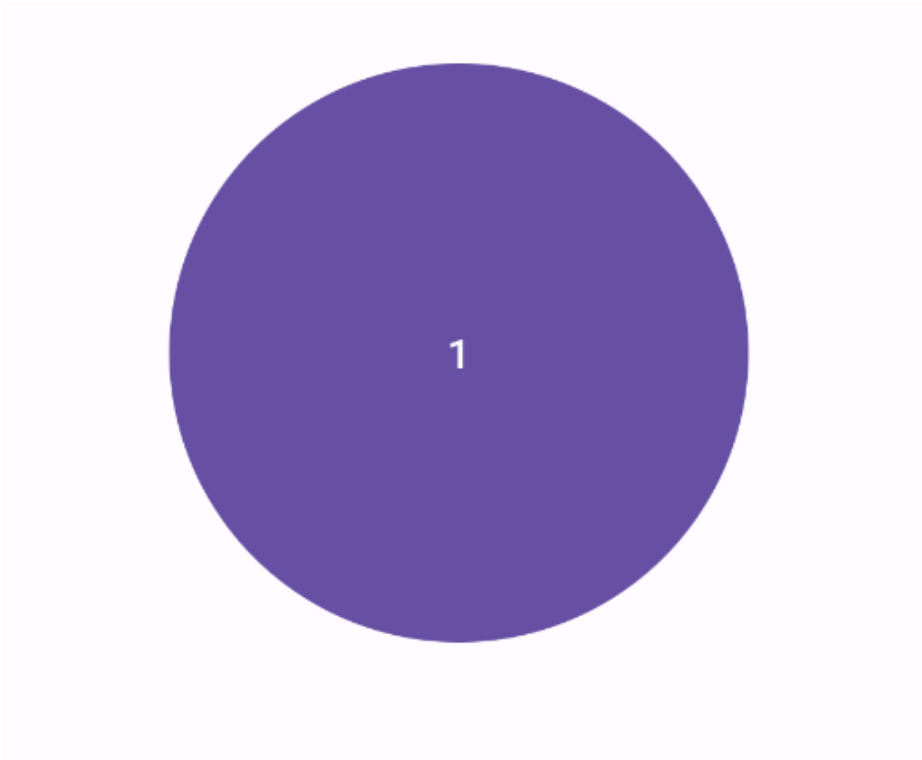




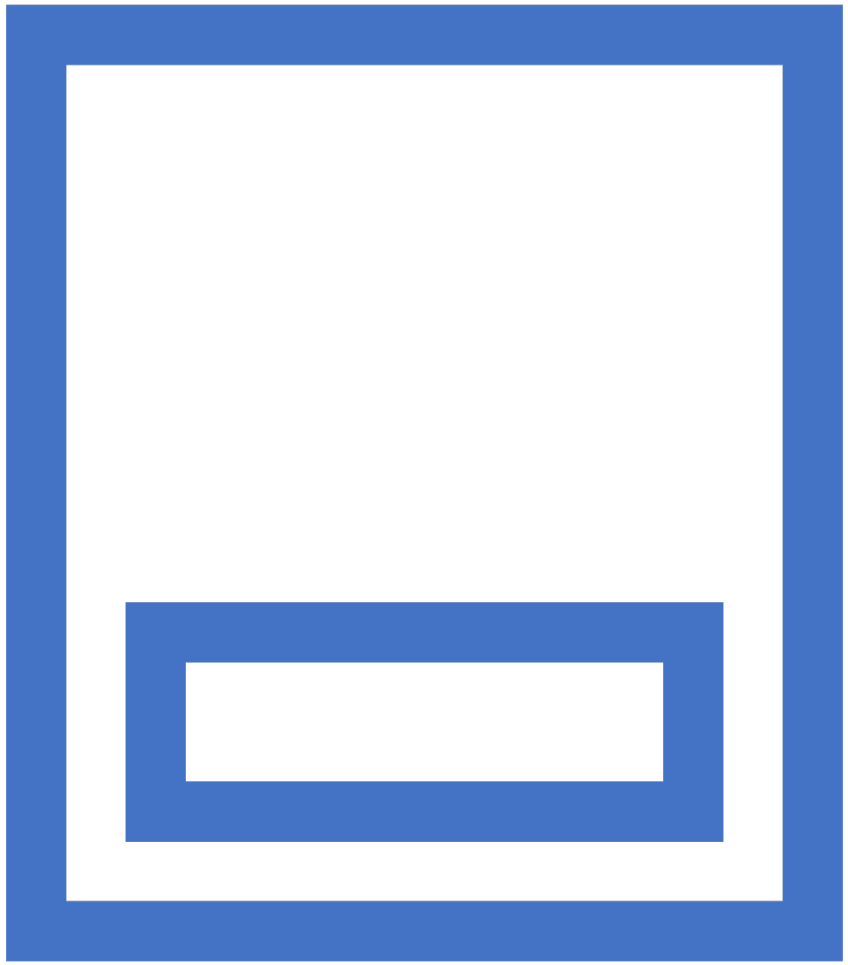
Round_button.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="oval">
    <solid android:color="@color/black"/>
    <size android:width="200dp"
        android:height="200dp"/>

</shape>
```

Using android:background



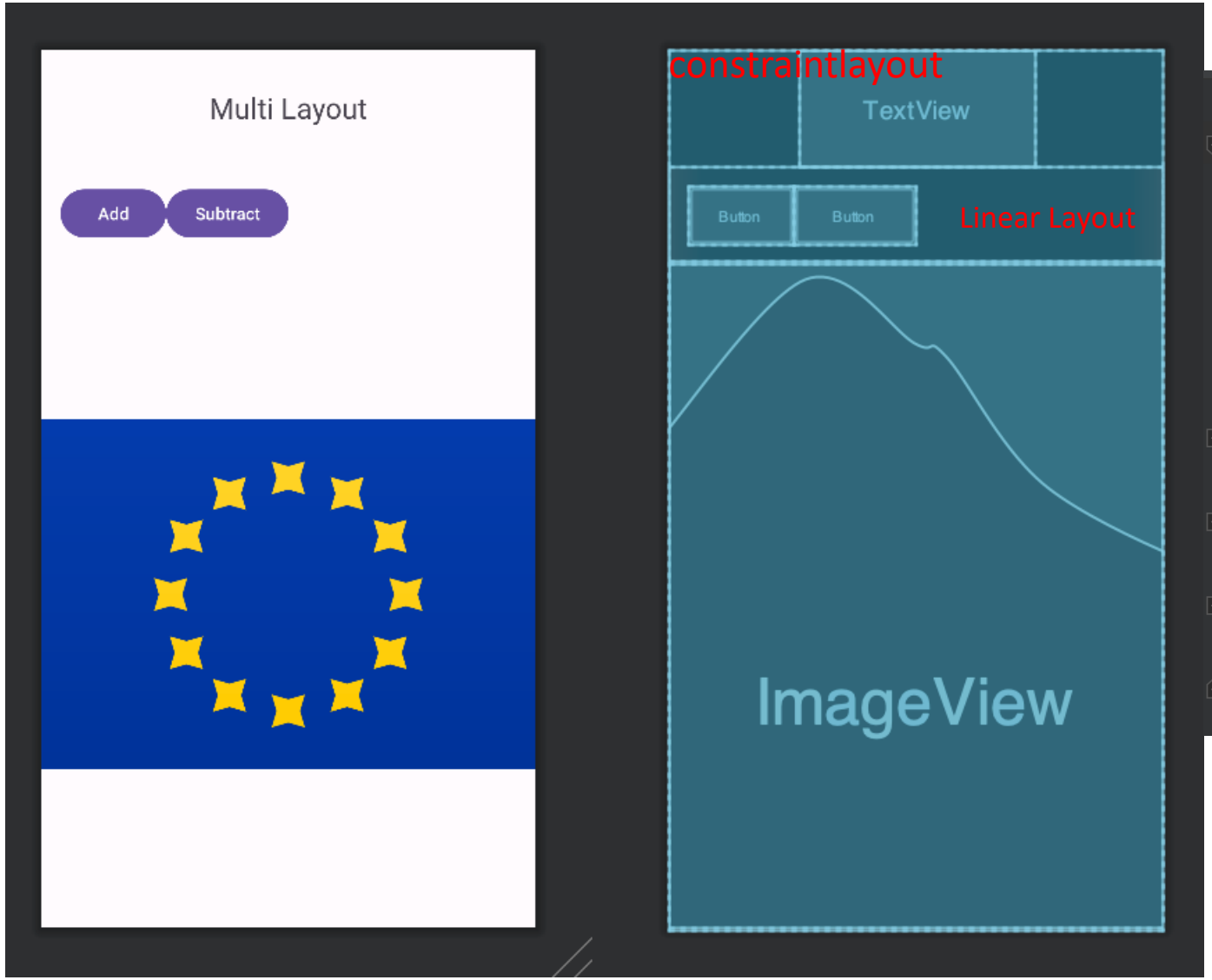

```
<Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="1"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/imageButton2"
    android:background="@drawable/round_button"/>
```



Nested view groups

- Nested view groups refer to the practice of placing one view group inside another view group to create complex and hierarchical user interfaces.
- This is a common approach when designing UIs that require more sophisticated layouts with multiple layers or sections.
- By nesting view groups, you can better organise and structure your UI elements, making it easier to manage and maintain the layout.

Example




```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns
    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=".NestedLayouts">

    <TextView...>

    <LinearLayout...>

    <ImageView...>

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

Example (Cont..)

```
    <LinearLayout
        android:id="@+id/linearLayoutButtons"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        app:layout_constraintTop_toBottomOf="@id/textViewTitle"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:padding="16dp">

        <Button
            android:id="@+id/button1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Add" />

        <Button
            android:id="@+id/button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Subtract" />
    </LinearLayout>

    <ImageView...>

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

All source codes are attached in the blackboard

Next week

Activities and passing data