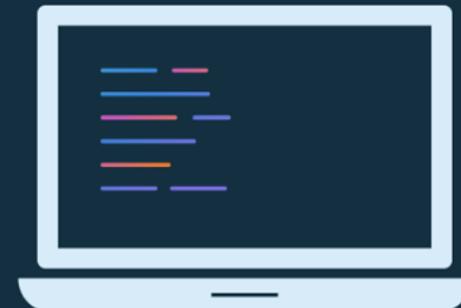
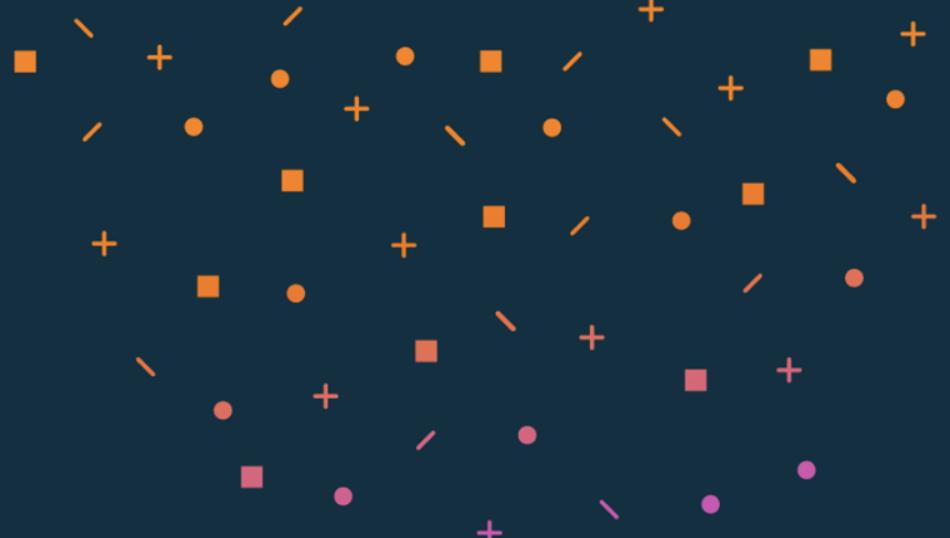




Android Development with Kotlin



Layouts and resources in Android

Everything you see is a View

If you look at your mobile device,
every user interface element that
you see is a **View**.



Views

- Views are the user interface building blocks in Android
 - Bounded by a rectangular area on the screen
 - Responsible for drawing and event handling
 - Examples: TextView, ImageView, Button
- Can be grouped to form more complex user interfaces

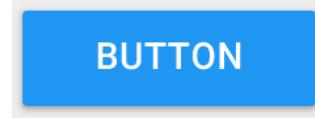
What is a View?

[View](#) is the base class for all visual interface elements

- commonly known as **controls** or **widgets**
- All UI controls, including the layout classes, are derived from View
 - Display text ([TextView](#) class), edit text ([EditText](#) class)
 - Buttons ([Button](#) class), [menus](#), and other controls
 - Scrollable ([ScrollView](#), [RecyclerView](#))
 - Show images ([ImageView](#))
 - Group views ([ConstraintLayout](#) and [LinearLayout](#))

Example of view subclasses

Button



EditText



Slider



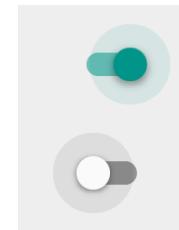
CheckBox



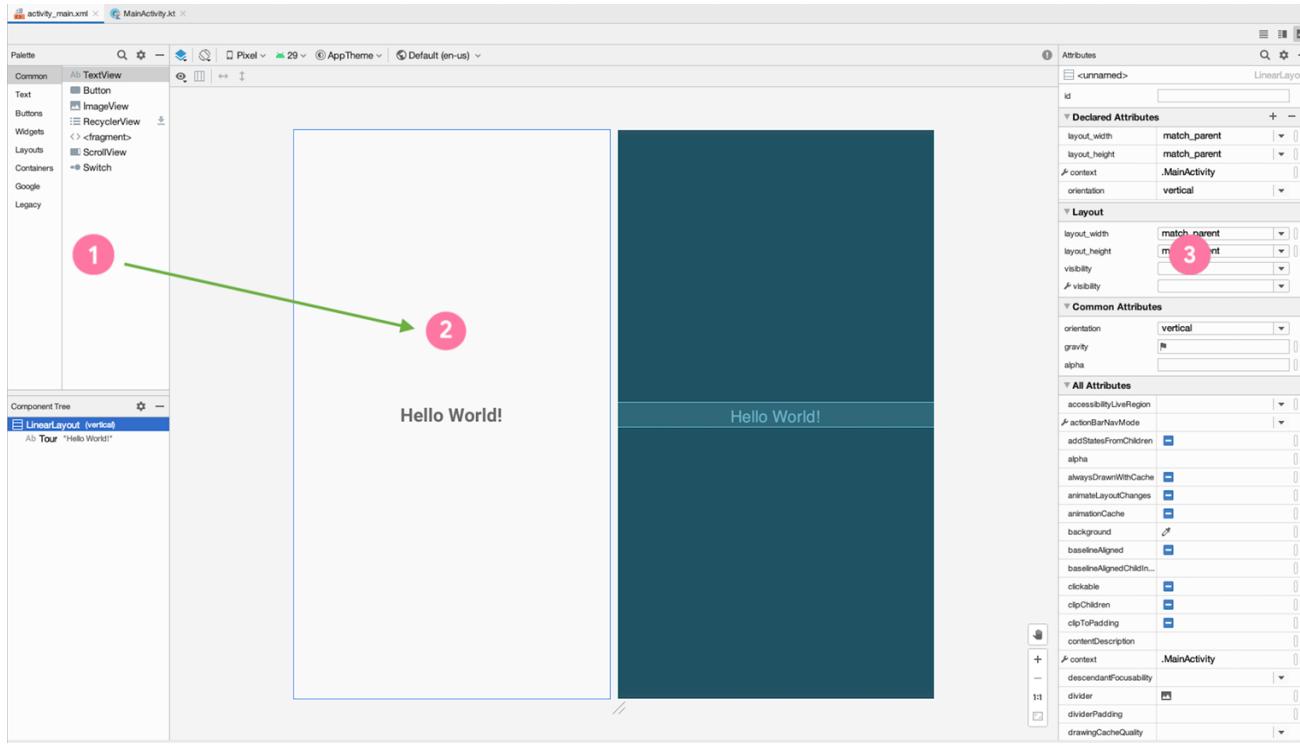
RadioButton



Switch



Layout Editor



XML Layouts

You can also edit your layout in XML.

- Android uses XML to specify the layout of user interfaces (including View attributes)
- Each View in XML corresponds to a class in Kotlin that controls how that View functions

XML for a TextView

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Hello World!"/>
```

Hello World!

Size of a View

- **wrap_content**

```
    android:layout_width="wrap_content"
```

- **match_parent**

```
    android:layout_width="match_parent"
```

- **Fixed value (use dp units)**

```
    android:layout_width="48dp"
```

ViewGroup contains "child" view

ViewGroups are extensions of the `View` class that can contain multiple child Views.

- [ConstraintLayout](#): Positions UI elements using constraint connections to other elements and to the layout edges
- [ScrollView](#): Contains one element and enables scrolling
- [RecyclerView](#): Contains a list of elements and enables scrolling by adding and removing elements dynamically

ViewGroup for layouts

Layouts

- are specific types of ViewGroups (subclasses of [ViewGroup](#))
- The most common way to define your layout and express the view hierarchy is with an XML layout file.
- contain child views
- can be in a row, column, grid, table, or absolute
- View objects are leaves in the tree,
ViewGroup objects are branches in the tree.

Common Layout Classes

- **ConstraintLayout:** Connect views with constraints
- **LinearLayout:** Horizontal or vertical row
- **RelativeLayout:** Child views relative to each other
- **TableLayout:** Rows and columns
- **FrameLayout:** Shows one child of a stack of children

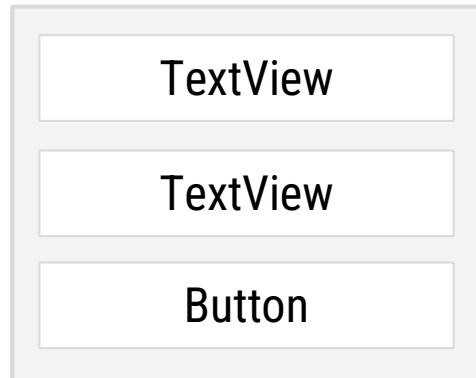
ViewGroups

A ViewGroup is a container that determines how views are displayed.

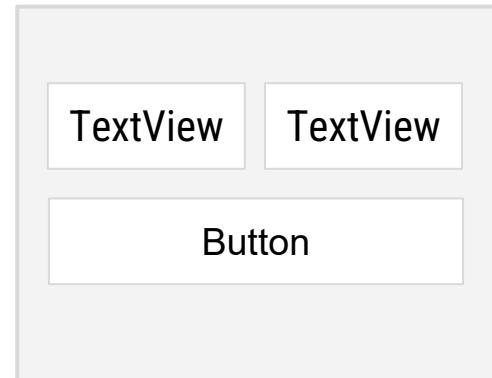
FrameLayout



LinearLayout



ConstraintLayout



The ViewGroup is the parent and the views inside it are its children.

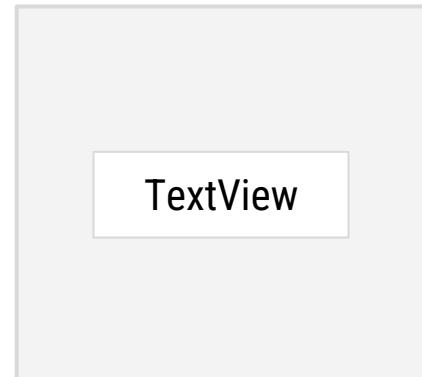
FrameLayout

- The simplest type of layout object.
- It's basically a blank space on your screen that you can later fill with a single object
- All child elements of the FrameLayout are pinned to the top left corner of the screen;
- you cannot specify a different location for a child view.
- Subsequent child views will simply be drawn over previous ones, partially or totally obscuring them (unless the newer object is transparent).

FrameLayout example

A FrameLayout generally holds a single child view.

```
<FrameLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
    <TextView  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:text="Hello World!" />  
</FrameLayout>
```

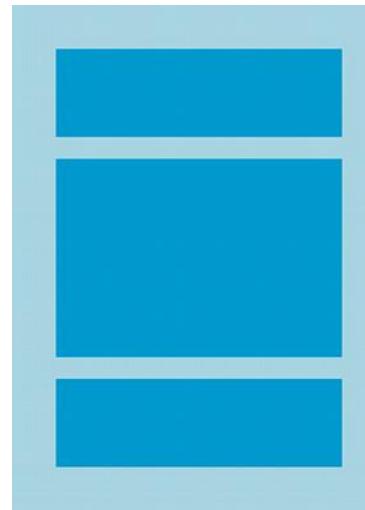
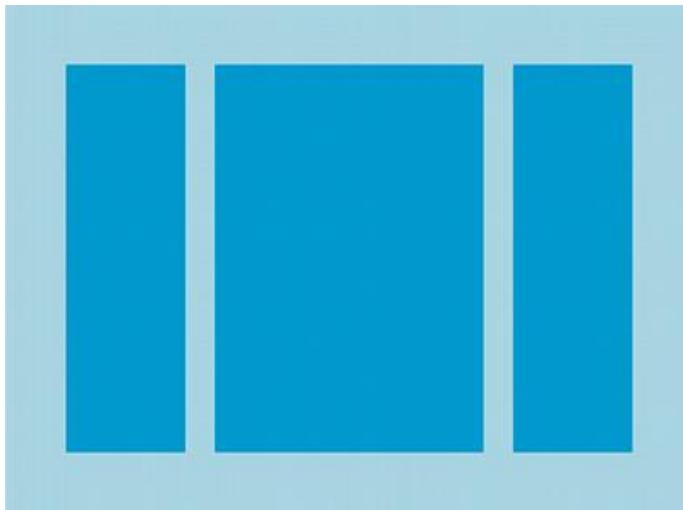


LinearLayout

- aligns all children in a single direction: depending on how you define the ***orientation*** attribute which is either ***horizontally*** or ***vertically***.
- All children are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding).
- It respects the ***margins*** between children and the ***gravity*** (right, center, or left alignment) of each child.

LinearLayout example

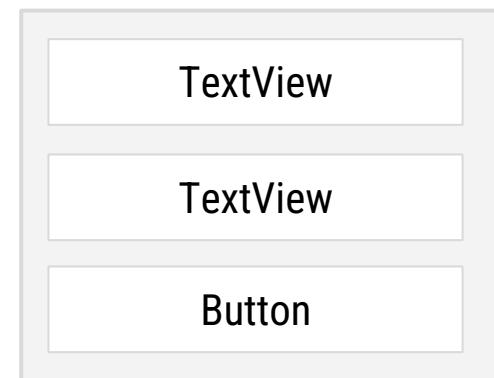
- Aligns child views in a row or column
- Set `android:orientation` to horizontal **or** vertical



LinearLayout example

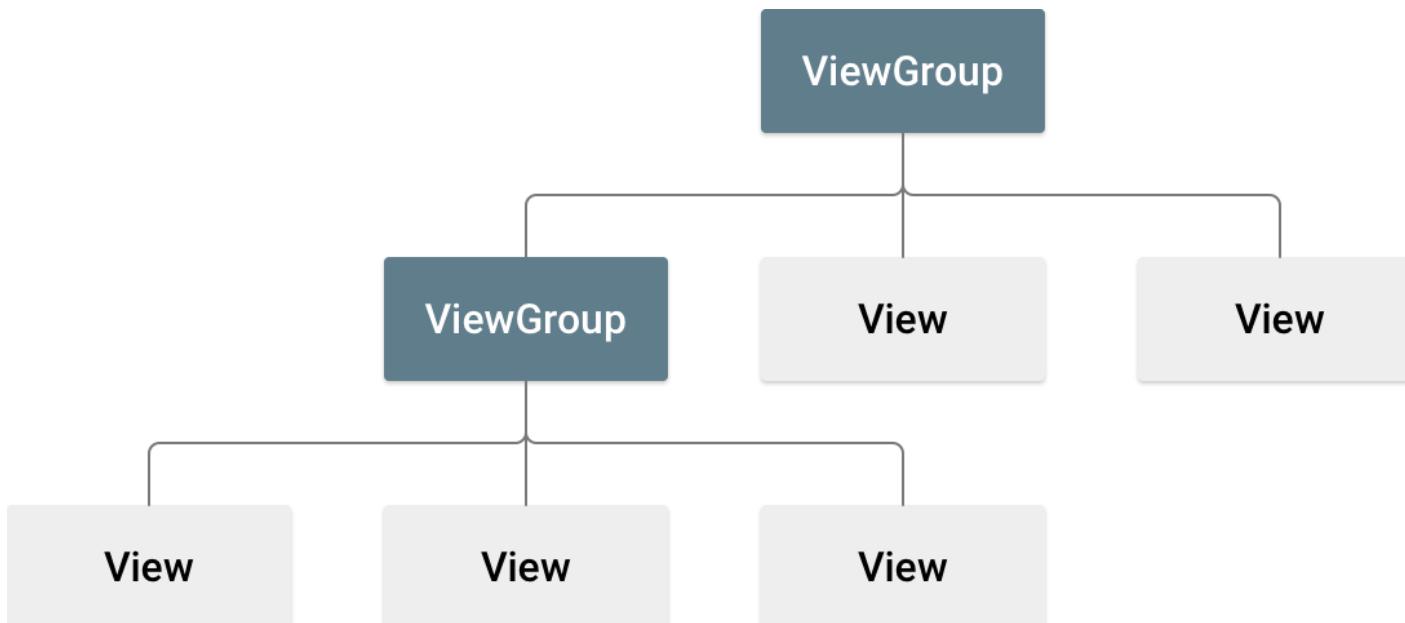
- Aligns child views in a row or column
- Set `android:orientation` to horizontal or vertical

```
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:orientation="vertical">  
    <TextView ... />  
    <TextView ... />  
    <Button ... />  
</LinearLayout>
```

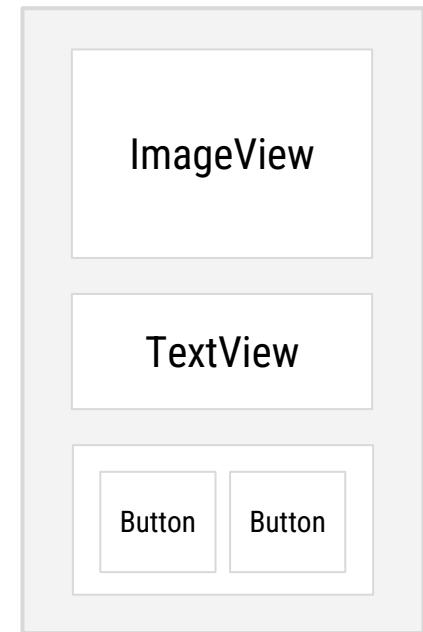
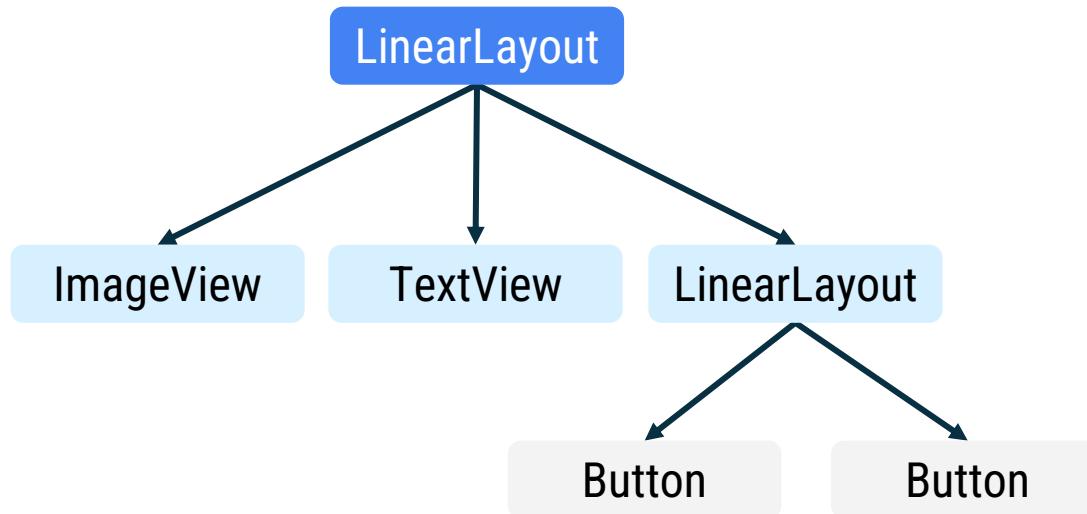


UI Structure

Root view is always a ViewGroup



View hierarchy



Defining the Layout

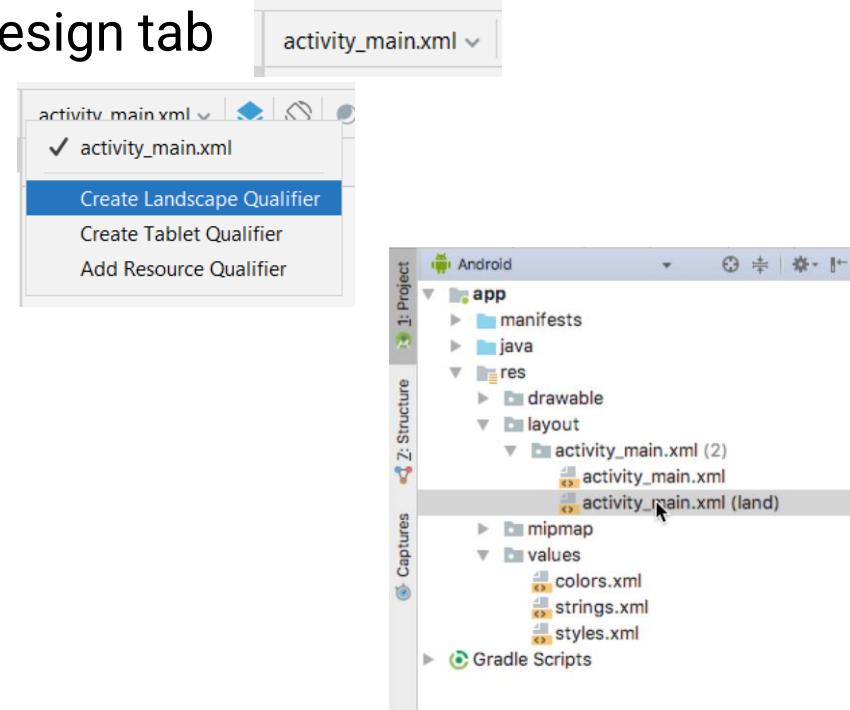
- Your layout is the architecture for the user interface in an Activity.
- It defines the layout structure and holds all the elements that appear to the user.
- You can declare your layout in two ways:
 - **Declare UI elements in XML**
 - **Instantiate layout elements at runtime.**

Advantages of XML resource

- It enables you to better separate the presentation of your application from the code that controls its behavior.
- Your UI descriptions are external to your application code, which means that you can modify or adapt it without having to modify your source code and recompile.
 - For example, you can create XML layouts for different screen orientations, different device screen sizes, and different languages.

Create layout variant for landscape

1. Click on the dropdown menu in the Design tab
2. Choose **Create Landscape Qualifier**
3. Layout variant created:
activity_main.xml (land)
4. Edit the layout variant as needed



Advantages of XML resource cont'd

- Declaring the layout in XML makes it easier to visualize the structure of your UI, so it's easier to debug problems.
- The XML vocabulary for declaring UI elements closely follows the structure and naming of the classes and methods, where **element names** correspond to **class names** and **attribute names** correspond to **methods**.

Advantages of XML resource cont'd

- In fact, the correspondence is often so direct that you can guess what XML attribute corresponds to a class method, or guess what class corresponds to a given XML element.
- However, note that not all vocabulary is identical.
In some cases, there are slight naming differences.
 - **For example**, the `EditText` element has a `text` attribute that corresponds to `EditText.setText()`.

Attributes

- Any View object may have an integer **ID** associated with it, to uniquely identify the View within the tree.
- When the application is compiled, this **ID** is referenced as an integer, but the **ID** is typically assigned in the layout XML file as a string, in the **id** attribute.
- The syntax for an **ID**, inside an XML tag is:

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

Attributes cont'd

- The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource.
- The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the **R.java** file).

App resources

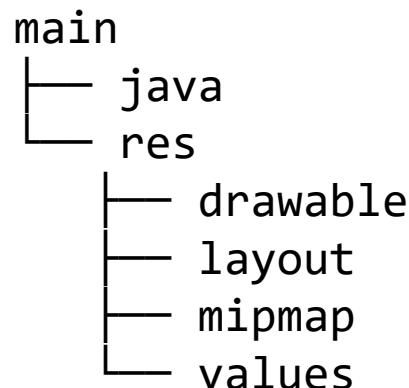
Static content or additional files that your code uses

- Layout files
- Images
- Audio files
- User interface strings
- App icon



Common resource directories

Add resources to your app by including them in the appropriate resource directory under the parent `res` folder.



Resource IDs

- Each resource has a resource ID to access it.
- When naming resources, the convention is to use all lowercase with underscores (for example, `activity_main.xml`).
- Android autogenerated a class file named `R.java` with references to all resources in the app.
- Individual items are referenced with:
`R.<resource_type>.<resource_name>`

Examples: `R.drawable.ic_launcher` (`res/drawable/ic_launcher.xml`)
`R.layout.activity_main` (`res/layout/activity_main.xml`)

Resource IDs for views

Individual views can also have resource IDs.

Add the `android:id` attribute to the View in XML. Use `@+id/name` syntax.

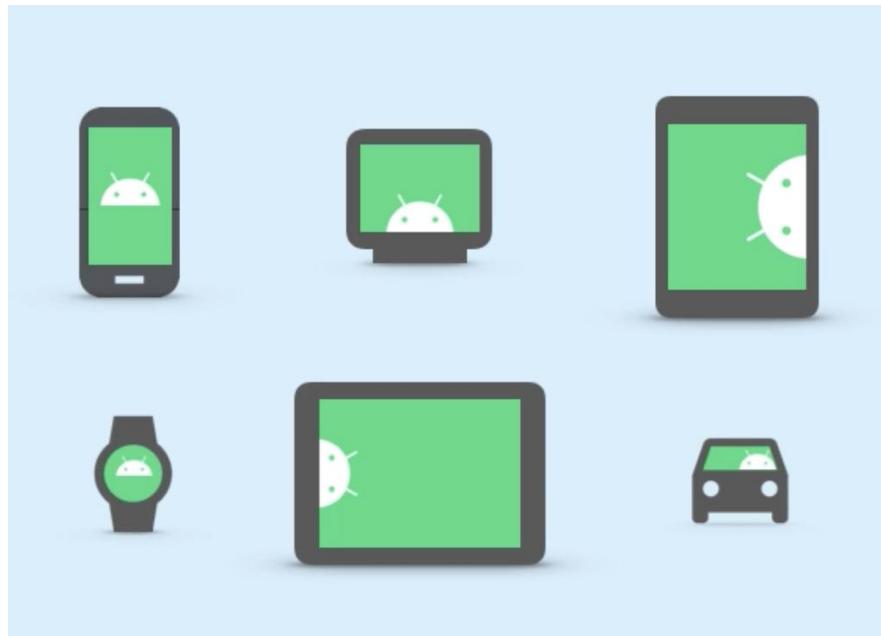
```
<TextView  
    android:id="@+id/helloTextView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Hello World!"/>
```

Within your app, you can now refer to this specific TextView using:
`R.id.helloTextView`

Layouts in Android

Android devices

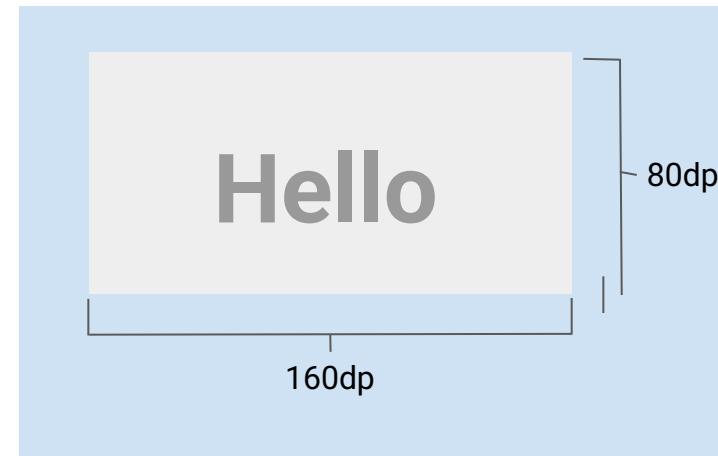
- Android devices come in many different form factors.
- More and more pixels per inch are being packed into device screens.
- Developers need the ability to specify layout dimensions that are consistent across devices.



Density-independent pixels (dp)

Use dp when specifying sizes in your layout, such as the width or height of views.

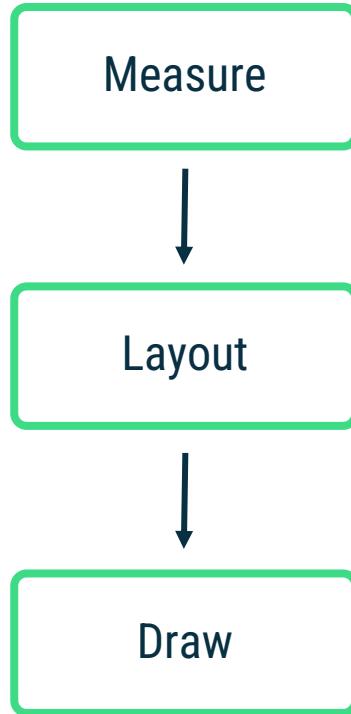
- Density-independent pixels (dp) take screen density into account.
- Android views are measured in density-independent pixels.
- $dp = \frac{\text{width in pixels} * 160}{\text{screen density}}$



Screen-density buckets

Density qualifier	Description	DPI estimate
ldpi (mostly unused)	Low density	~120dpi
mdpi (baseline density)	Medium density	~160dpi
hdpi	High density	~240dpi
xhdpi	Extra-high density	~320dpi
xxhdpi	Extra-extra-high density	~480dpi
xxxhdpi	Extra-extra-extra-high density	~640dpi

Android View rendering cycle



Drawing region

What we see:

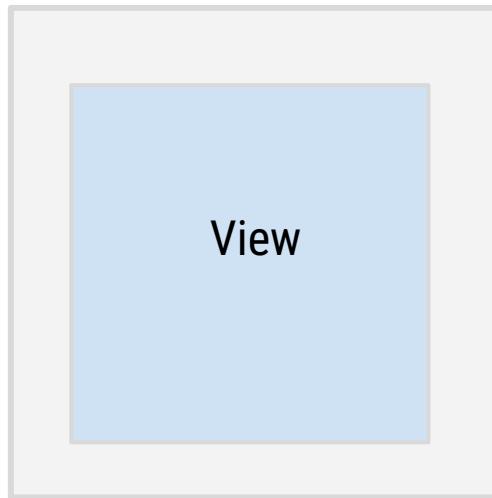


How it's drawn:

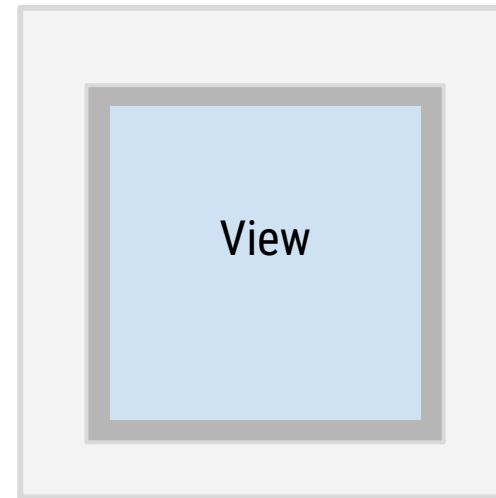


View margins and padding

View with margin



View with margin and padding



ConstraintLayout

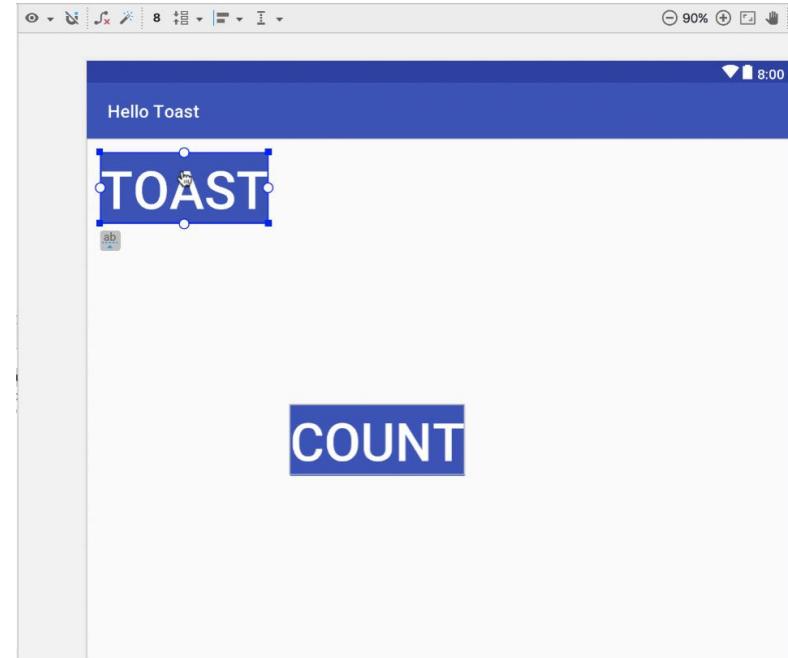
Deeply nested layouts are costly

- Deeply nested ViewGroups require more computation
- Views may be measured multiple times
- Can cause UI slowdown and lack of responsiveness

Use ConstraintLayout to avoid some of these issues!

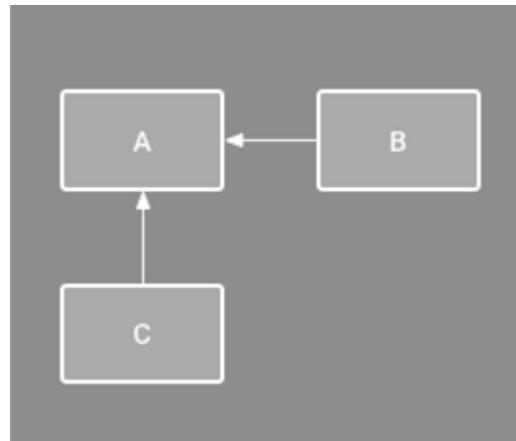
What is ConstraintLayout?

- Recommended default layout for Android
- Solves costly issue
of too many nested layouts,
while allowing complex behavior
- Position and size views within it
using a set of constraints



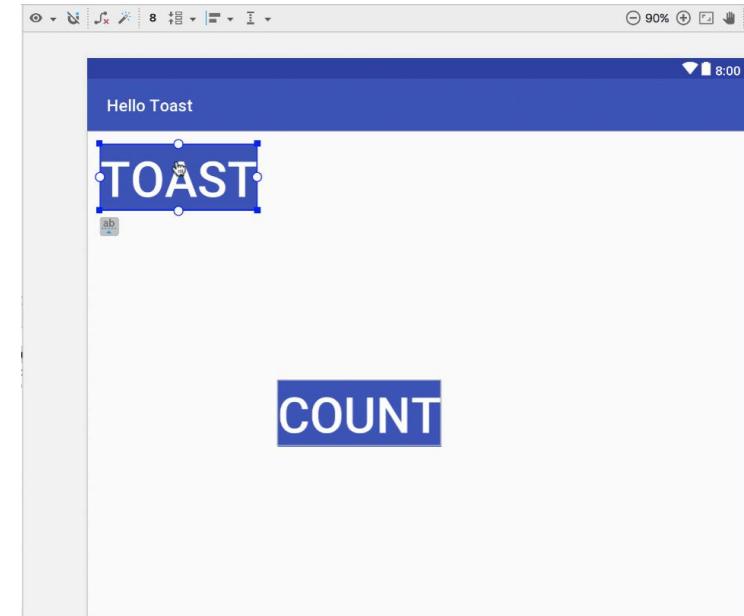
What is a constraint?

A restriction or limitation on the properties of a View that the layout attempts to respect



The layout editor with ConstraintLayout

- Connect UI elements to the parent layout
- Resize and position elements
- Align elements with others
- Adjust margins and dimensions
- Change attributes



Relative positioning constraints

Can set up a constraint relative to the parent container

Format:

```
layout_constraint<SourceConstraint>_to<TargetConstraint>of
```

Example attributes on a TextView:

```
app:layout_constraintTop_toTopOf="parent"
```

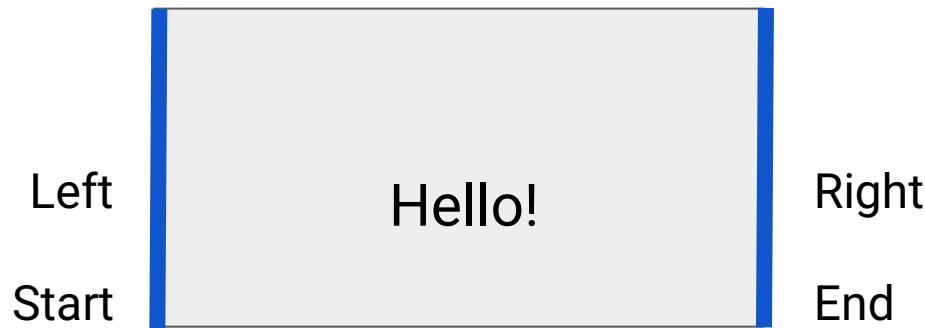
```
app:layout_constraintLeft_toLeftOf="parent"
```



Relative positioning constraints

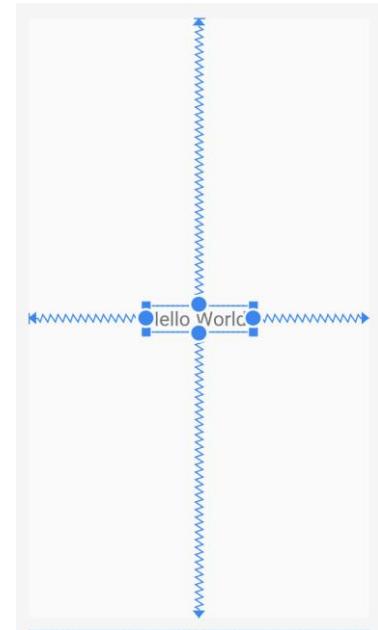


Relative positioning constraints



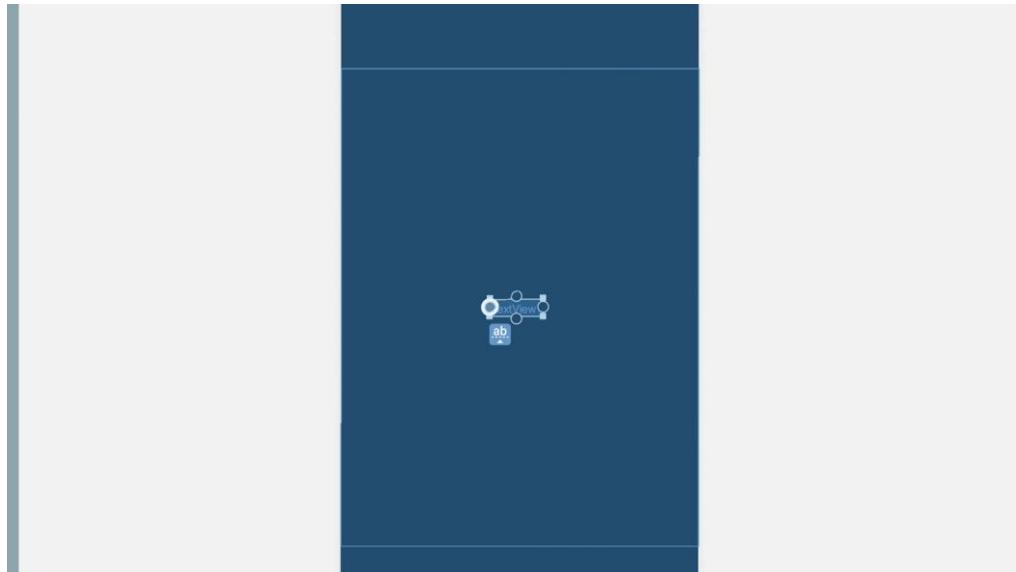
Simple ConstraintLayout example

```
<androidx.constraintlayout.widget.ConstraintLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <TextView  
        ...  
  
        app:layout_constraintBottom_toBottomOf="parent"  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintStart_toStartOf="parent"  
        app:layout_constraintTop_toTopOf="parent" />  
    </androidx.constraintlayout.widget.ConstraintLayout>
```



Layout Editor in Android Studio

You can click and drag to add constraints to a View.



Constraint Widget in Layout Editor



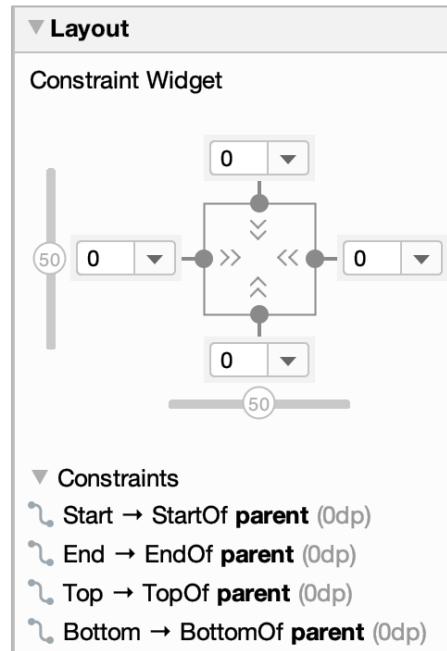
Fixed



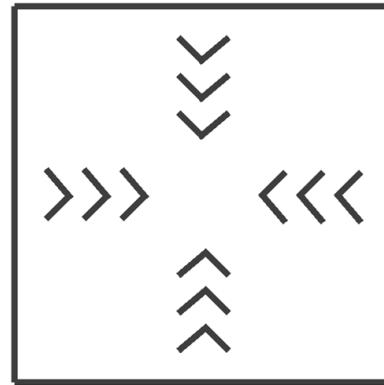
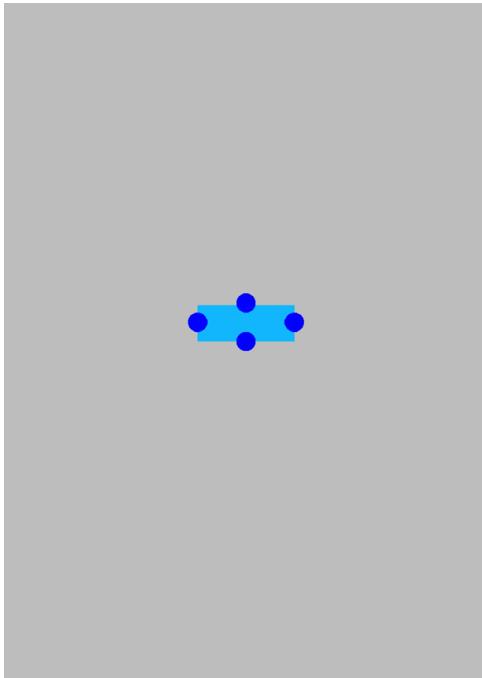
Wrap content



Match constraints



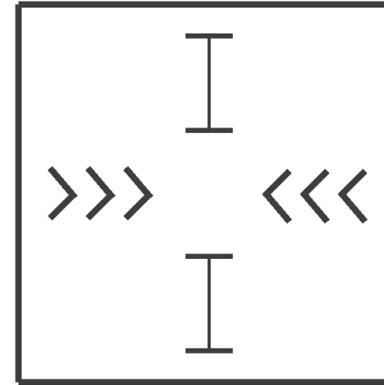
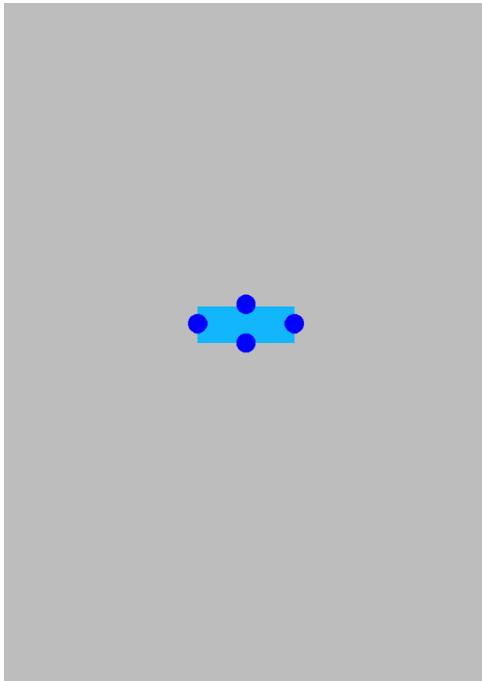
Wrap content for width and height



`layout_width` `wrap_content`

`layout_height` `wrap_content`

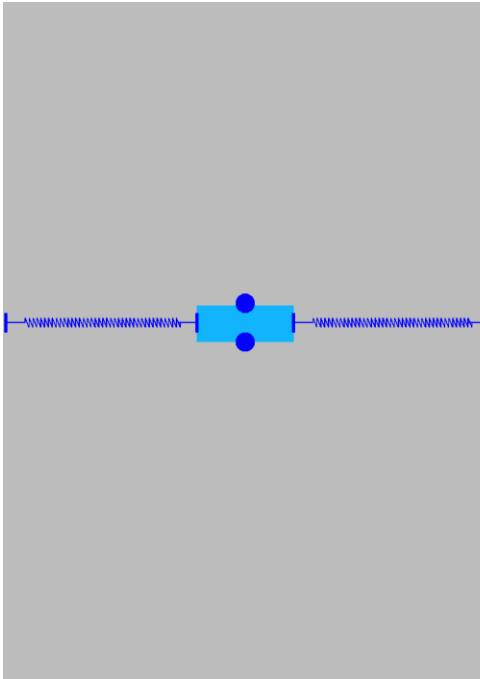
Wrap content for width, fixed height



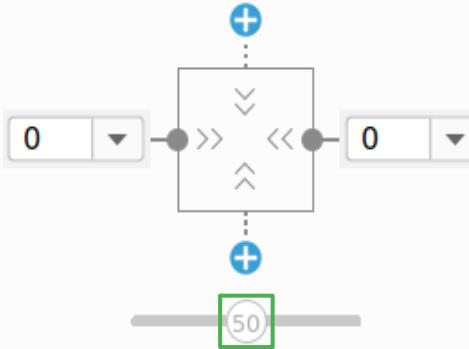
layout_width **wrap_content**

layout_height 48dp

Center a view horizontally



Constraint Widget

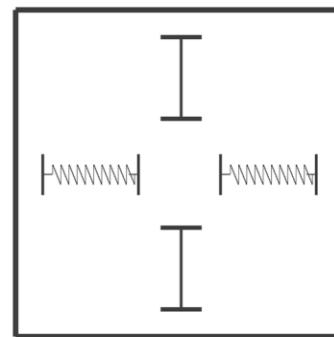
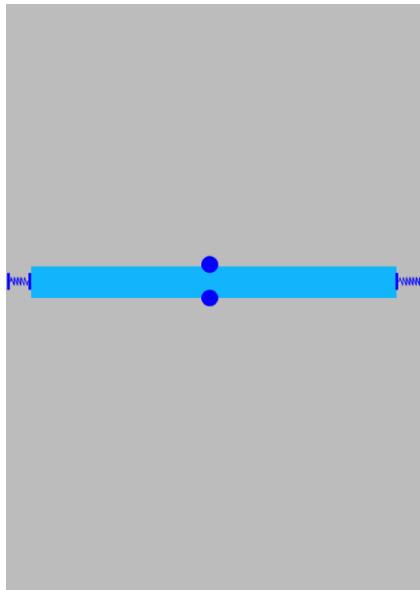


▼ Constraints

- ↳ Left → LeftOf **parent** (0dp)
- ↳ Right → RightOf **parent** (0dp)

Use match_constraint

Can't use `match_parent` on a child view, use `match_constraint` instead



`layout_width 0dp(match_constraint)`

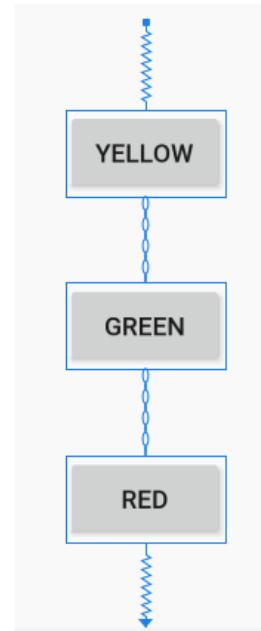
`layout_height 48dp`

Chains

- Let you position views in relation to each other
- Can be linked horizontally or vertically
- Provide much of LinearLayout functionality

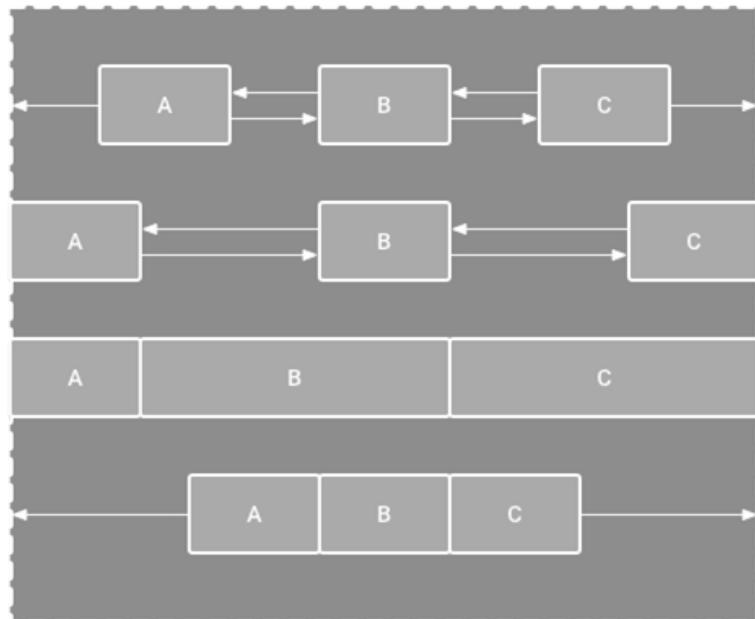
Create a Chain in Layout Editor

1. Select the objects you want to be in the chain.
2. Right-click and select **Chains**.
3. Create a horizontal or vertical chain.



Chain styles

Adjust space between views with these different chain styles.



Spread Chain

Spread Inside Chain

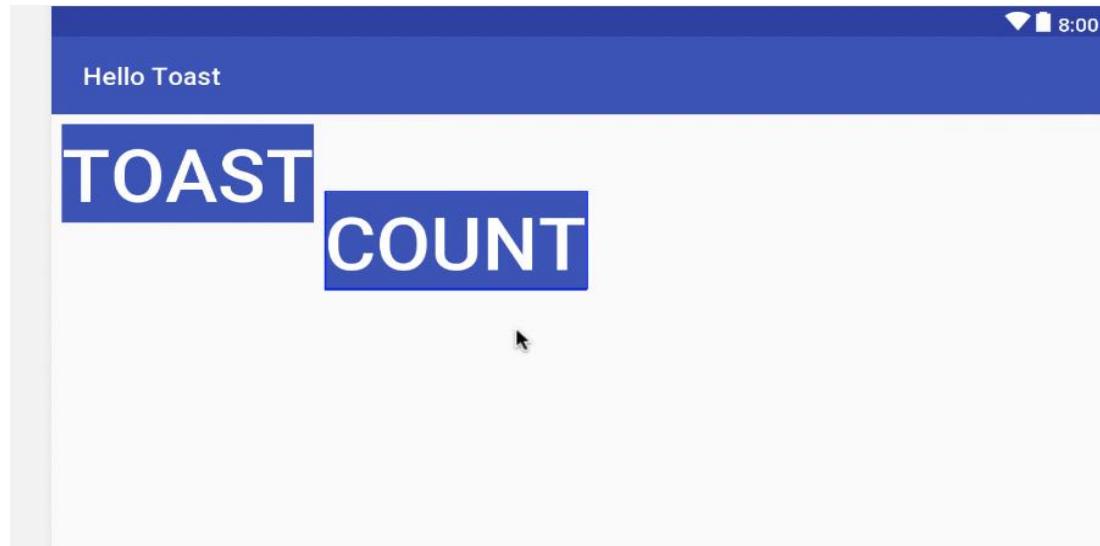
Weighted Chain

Packed Chain

Additional topics for ConstraintLayout

Align elements by baseline

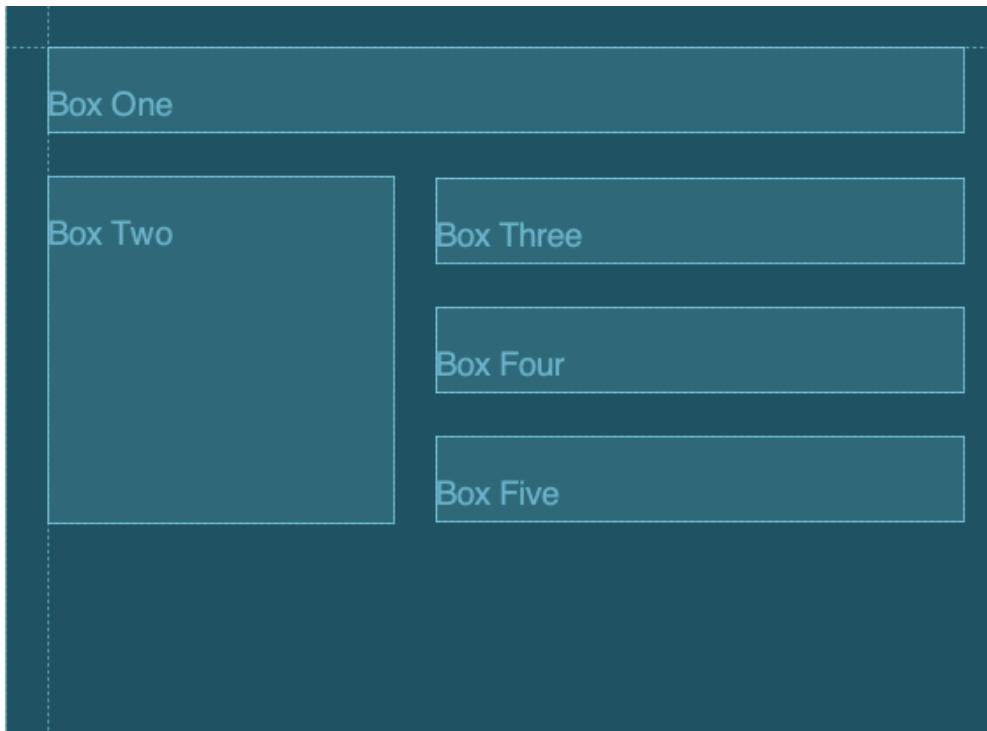
1. Click the  baseline constraint button
2. Drag from baseline
to another element's
baseline



Guidelines

- Let you position multiple views relative to a single guide
- Can be vertical or horizontal
- Allow for greater collaboration with design/UX teams
- Aren't drawn on the device

Guidelines in Android Studio



Example Guideline

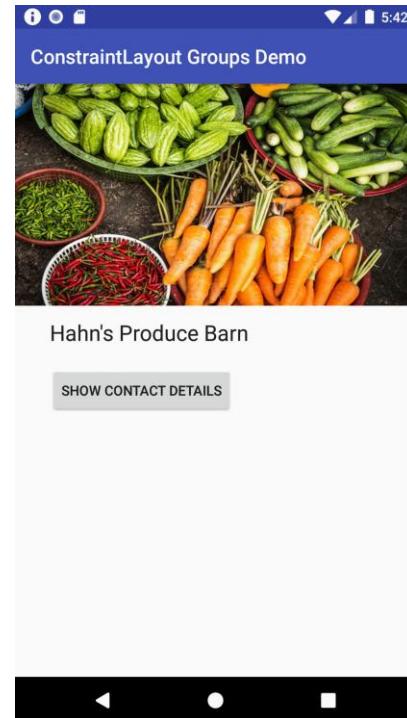
```
<ConstraintLayout>
    <androidx.constraintlayout.widget.Guideline
        android:id="@+id/start_guideline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        app:layout_constraintGuide_begin="16dp" />
    <TextView ...
        app:layout_constraintStart_toEndOf="@+id/start_guideline" />
</ConstraintLayout>
```

Creating Guidelines

- `layout_constraintGuide_begin`
- `layout_constraintGuide_end`
- `layout_constraintGuide_percent`

Groups

- Control the visibility of a set of widgets
- Group visibility can be toggled in code



Example group

```
<androidx.constraintlayout.widget.Group  
    android:id="@+id/group"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:constraint_referenced_ids="locationLabel,locationDetails"/>
```

Groups app code

```
override fun onClick(v: View?) {  
    if (group.visibility == View.GONE) {  
        group.visibility = View.VISIBLE  
        button.setText(R.string.hide_details)  
    } else {  
        group.visibility = View.GONE  
        button.setText(R.string.show_details)  
    }  
}
```

Accessibility

Accessibility

- Refers to improving the design and functionality of your app to make it easier for more people, including those with disabilities, to use
- Making your app more accessible leads to an overall better user experience and benefits all your users

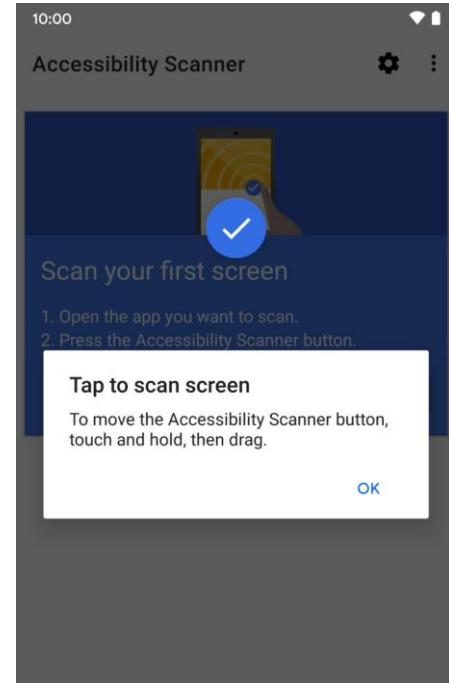
Make apps more accessible

- Increase text visibility with foreground and background color contrast ratio:
 - At least 4.5:1 for small text against the background
 - At least 3.0:1 for large text against the background
- Use large, simple controls
 - Touch target size should be at least 48dp x 48dp
- Describe each UI element
 - Set content description on images and controls

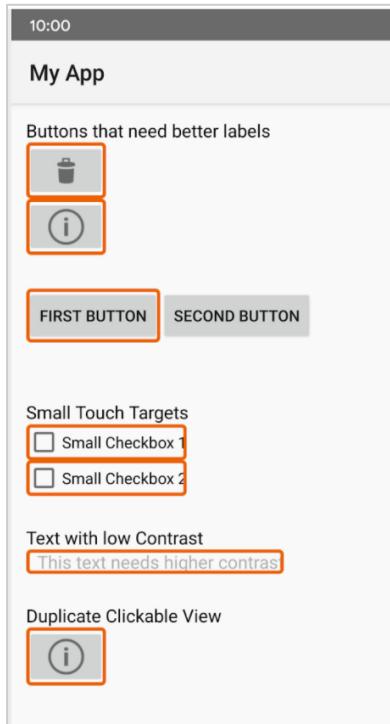
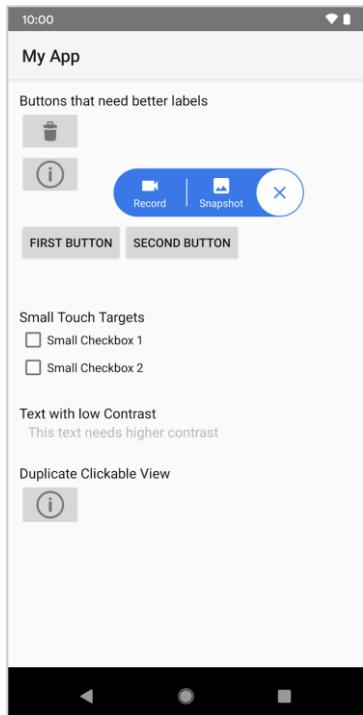
Accessibility Scanner

Tool that scans your screen and suggests improvements to make your app more accessible, based on:

- Content labels
- Touch target sizes
- Clickable views
- Text and image contrast



Accessibility Scanner example



This screenshot shows a detailed view of an accessibility issue for 'Element 6'. It highlights 'Text with low Contrast' (the text 'This text needs higher contrast'). A tooltip provides a suggestion: 'Consider increasing this item's text foreground to background contrast ratio.' Below this, it shows a contrast ratio of 1.90 and a suggested ratio of 4.50. It also lists the foreground color (#B8B8B8) and background color (#FAFAFA).

Add content labels

- Set `contentDescription` attribute → read aloud by screen reader

```
<ImageView  
    ...  
    android:contentDescription="@string/stop_sign" />
```

- Text in `TextView` already provided to accessibility services, no additional label needed

No content label needed

- For graphical elements that are purely for decorative purposes, you can set

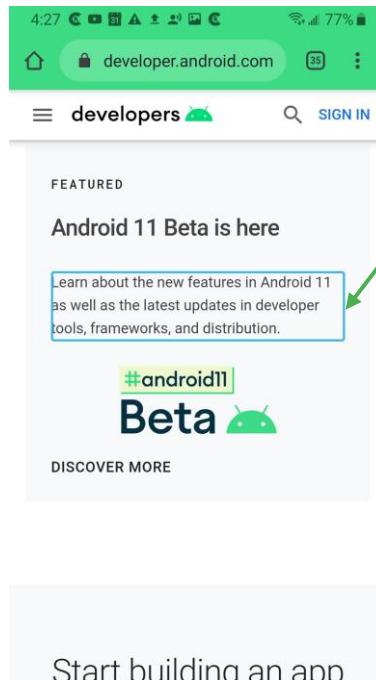
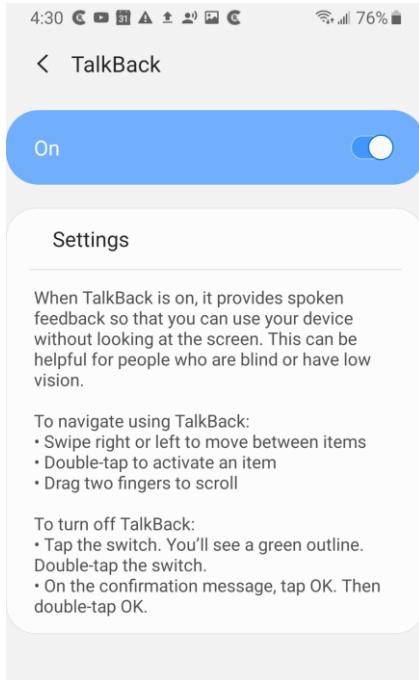
```
android:importantForAccessibility="no"
```

- Removing unnecessary announcements is better for the user

TalkBack

- Google screen reader included on Android devices
- Provides spoken feedback so you don't have to look at the screen to use your device
- Lets you navigate the device using gestures
- Includes braille keyboard for Unified English Braille

TalkBack example



Reads text
aloud as user
navigates the
screen

Start building an app

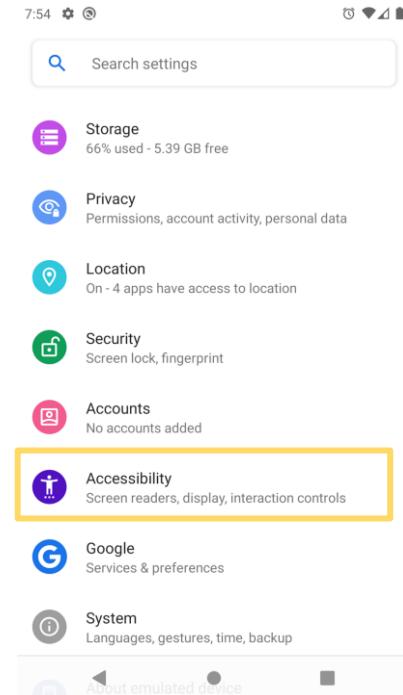
Switch access

- Allows for controlling the device using one or more switches instead of the touchscreen
- Scans your app UI and highlights each item until you make a selection
- Use with external switch, external keyboard, or buttons on the Android device (e.g., volume buttons)

Android Accessibility Suite

Collection of accessibility apps that help you use your Android device eyes-free, or with a switch device. It includes:

- Talkback screen reader
- Switch Access
- Accessibility Menu
- Select to Speak



Accessibility Resources

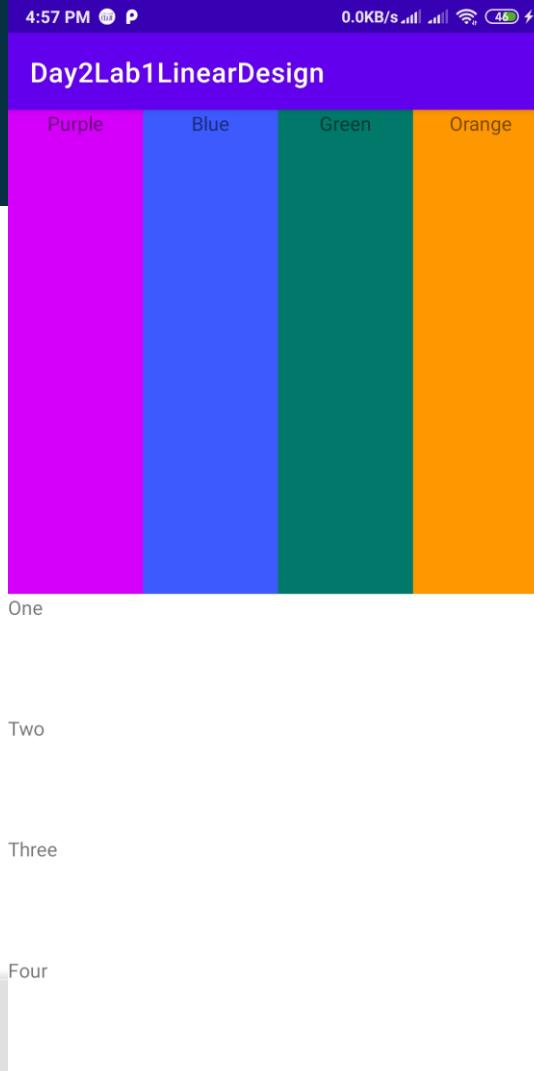
- [Build more accessible apps](#)
- [Principles for improving app accessibility](#)
- [Basic Android Accessibility codelab](#)
- [Material Design best practices on accessibility](#)

Demo

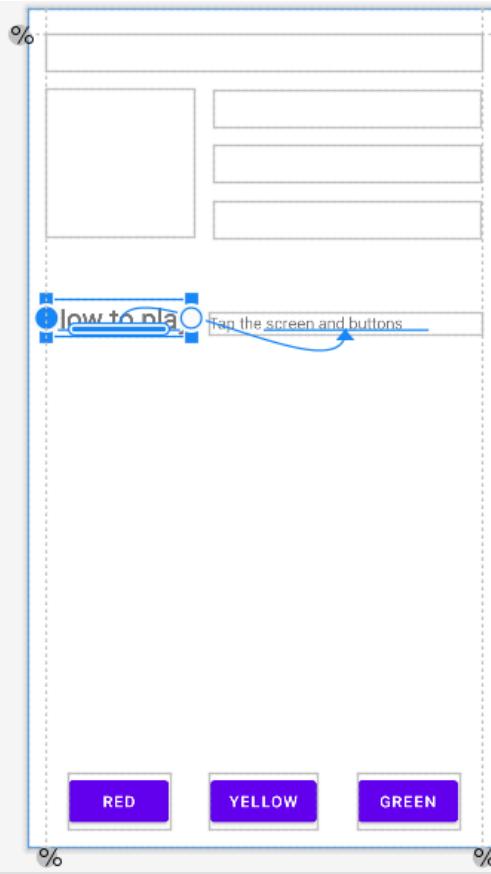


Assignment 1

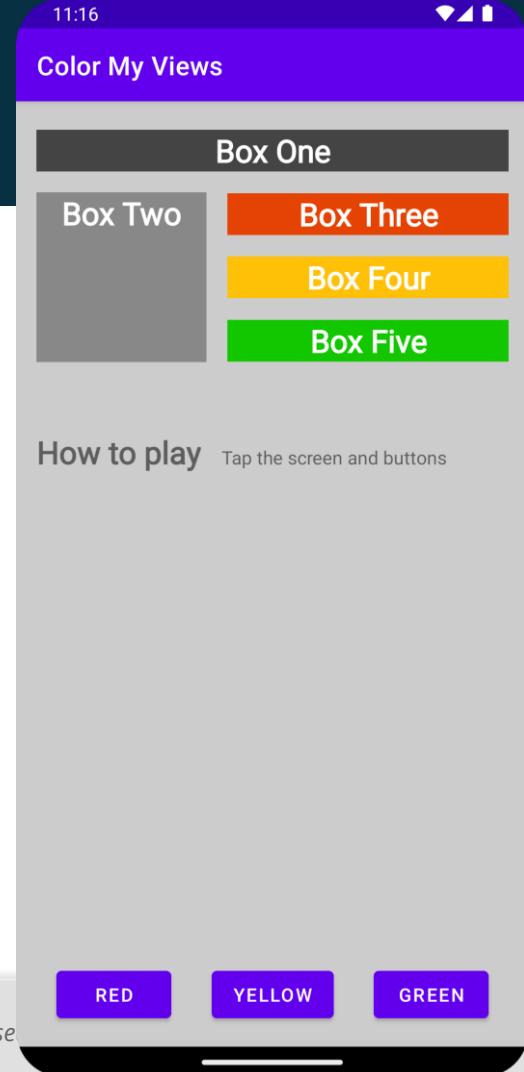
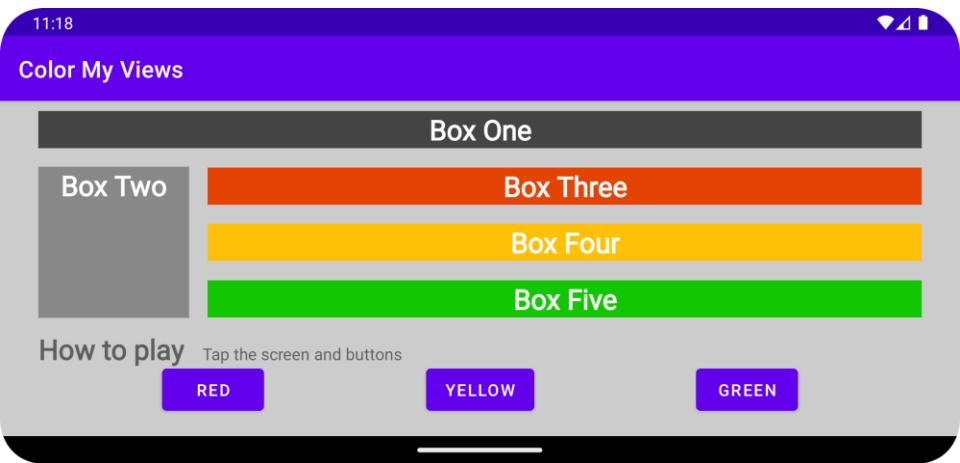
Draw this UI using LinearLayout only



Assignment 2



Assignment 2



Assignment 3

[Basic Android Accessibility codelab](#)