BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

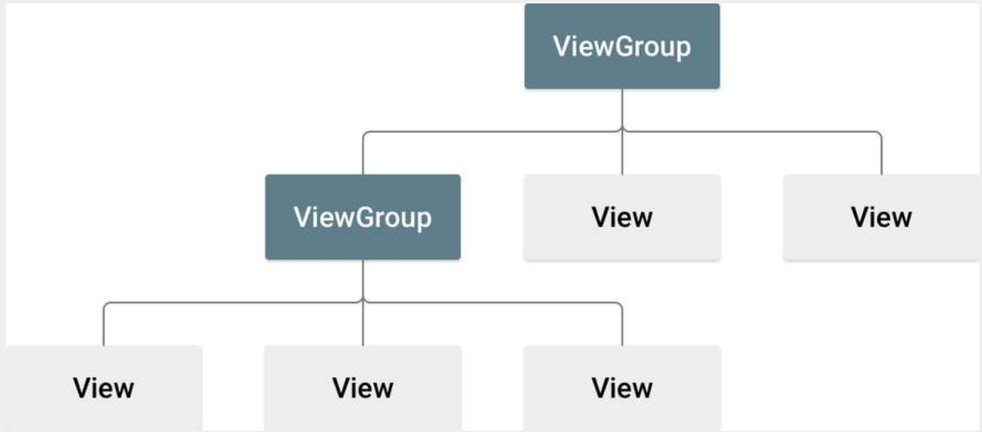
Mobile development and security

Session 3

Karim Karimov
Lecturer

Layouts

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of **View** and **ViewGroup** objects. A View usually draws something the user can see and interact with. Whereas a **ViewGroup** is an invisible container that defines the layout structure for **View** and other **ViewGroup** objects, as shown in figure below.



Layouts

The **View** objects are usually called ***"widgets"*** and can be one of many subclasses, such as **Button** or **TextView**. The **ViewGroup** objects are usually called ***"layouts"*** can be one of many types that provide a different layout structure, such as **LinearLayout** or **ConstraintLayout** .
You can declare a layout in three ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.
- You can also use **Android Studio's Layout Editor** to build your XML layout using a drag-and-drop interface.
- **Instantiate layout elements at runtime**. Your app can create **View** and **ViewGroup** objects (and manipulate their properties) programmatically.
- **Jetpack Compose.** Your app build UI content using new Compose library with Kotlin.

Layouts

Declaring your UI in XML allows you to separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for **different screen sizes and orientations**.
The Android framework gives you the flexibility to use either or both of these methods to build your app's UI. For example, you can declare your app's default layouts in XML, and then modify the layout at runtime.

Layouts with XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a **View** or **ViewGroup** object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical **LinearLayout** to hold a **TextView** and a **Button** (next slide).

After you've declared your layout in XML, save the file with the **.xml** extension, in your Android project's **res/layout/** directory, so it will properly compile.

Layouts with 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:orientation="vertical" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Load the XML Resource

When you compile your app, each XML layout file is compiled into a View resource. You should load the layout resource from your app code, in your **Activity.onCreate()** callback implementation. Do so by calling **setContentView()**, passing it the reference to your layout resource in the form of: **R.layout.layout_file_name**. For example, if your XML layout is saved as main_layout.xml, you would load it for your Activity like below:

```
fun onCreate(savedInstanceState: Bundle) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.main_layout)
}
```

Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, **TextView** supports the ***textSize*** attribute), but these attributes are also inherited by any **View** objects that may extend this class. Some are common to all **View** objects, because they are inherited from the root **View** class (like the ***id*** attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the **View** object, as defined by that object's parent **ViewGroup** object.

ID attribute

Any View object may have an **integer ID** associated with it, to uniquely identify the View within the tree. When the app 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. This is an XML attribute common to all **View** objects (defined by the **View** class) and you will use it very often. The syntax for an ID, inside an XML tag is:

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

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).

ID attribute

In order to create views and reference them from the app, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```

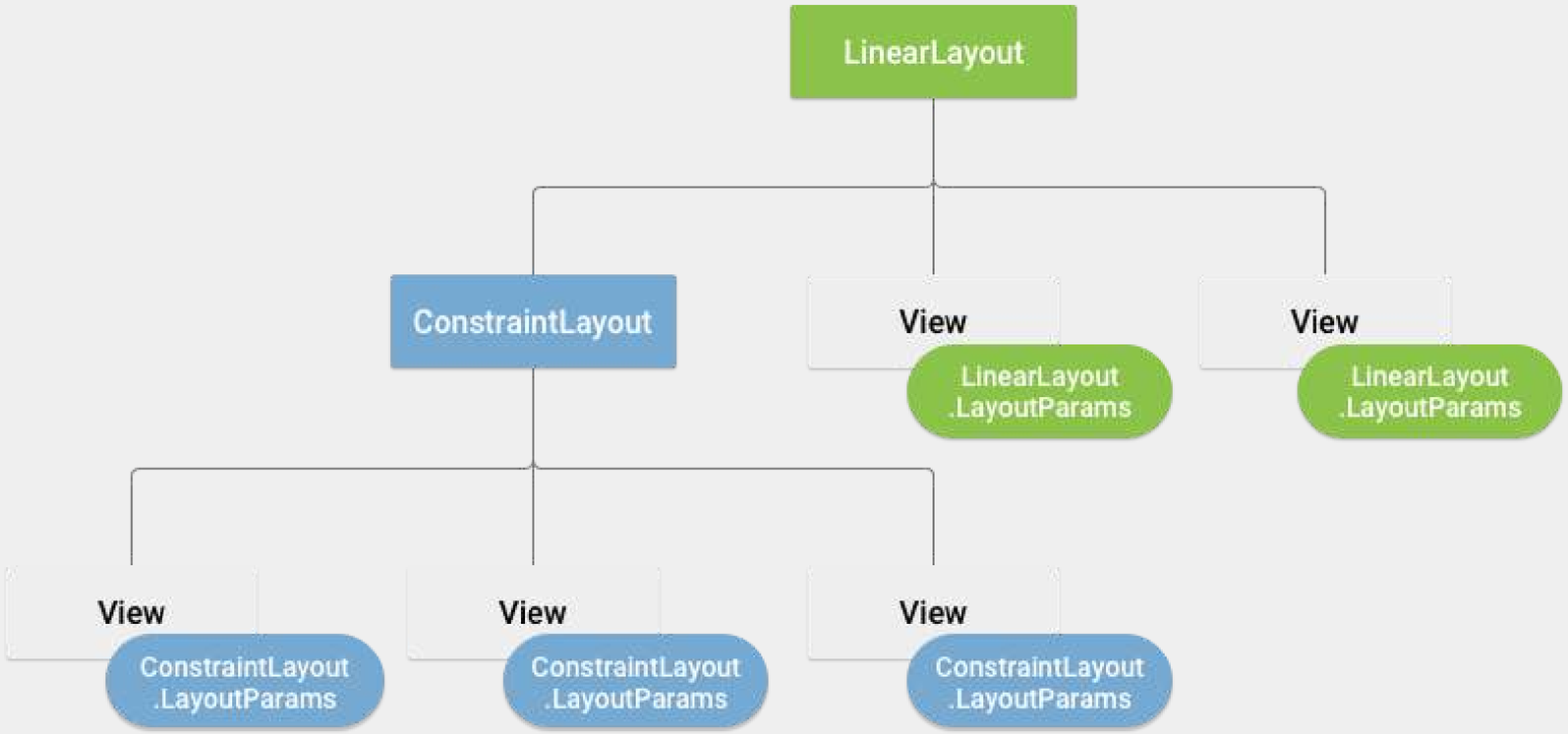
2. Then create an instance of the view object and capture it from the layout (typically in the onCreate() method):

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

Layout parameters

XML layout attributes named ***layout_something*** define layout parameters for the **View** that are appropriate for the **ViewGroup** in which it resides.
Every **ViewGroup** class implements a nested class that extends **ViewGroup.LayoutParams**. This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in figure (next slide), the parent view group defines layout parameters for each child view (including the child view group).

Layout parameters



Layout parameters

Note that every **LayoutParams** subclass has its own syntax for setting values. Each child element must define **LayoutParams** that are appropriate for its parent, though it may also define different **LayoutParams** for its own children.
All view groups include a width and height (**layout_width** and **layout_height**), and each view is required to define them. Many **LayoutParams** also include optional margins and borders.

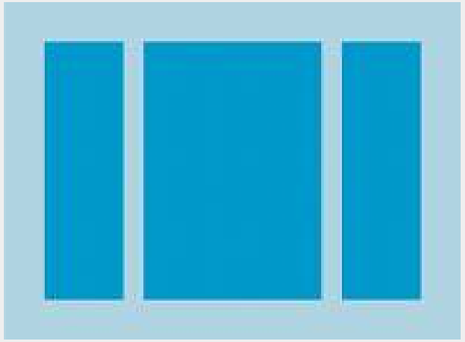
You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

- **wrap_content** tells your view to size itself to the dimensions required by its content.
- **match_parent** tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (**dp**), **wrap_content**, or **match_parent**, is a better approach, because it helps ensure that your app will display properly across a variety of device screen sizes.

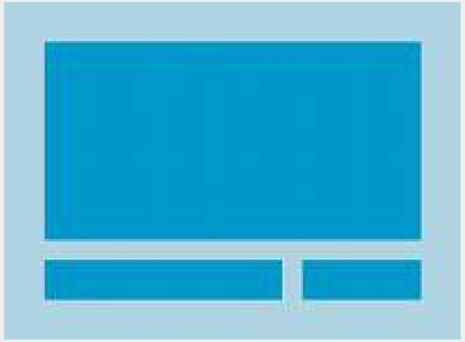
Common layouts

Linear layout



A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.

Relative layout



Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).

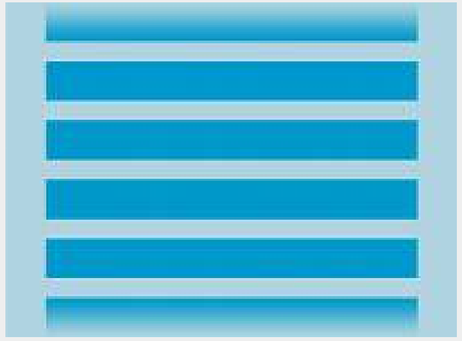
Web layout

Displays web pages.

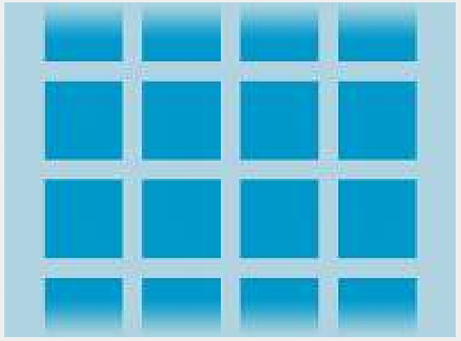
Building Layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses **AdapterView** to populate the layout with views at runtime. A subclass of the AdapterView class uses an **Adapter** to bind data to its layout. The Adapter behaves as a middleman between the data source and the AdapterView layout—the Adapter retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the AdapterView layout.

List view



Grid view



UI components

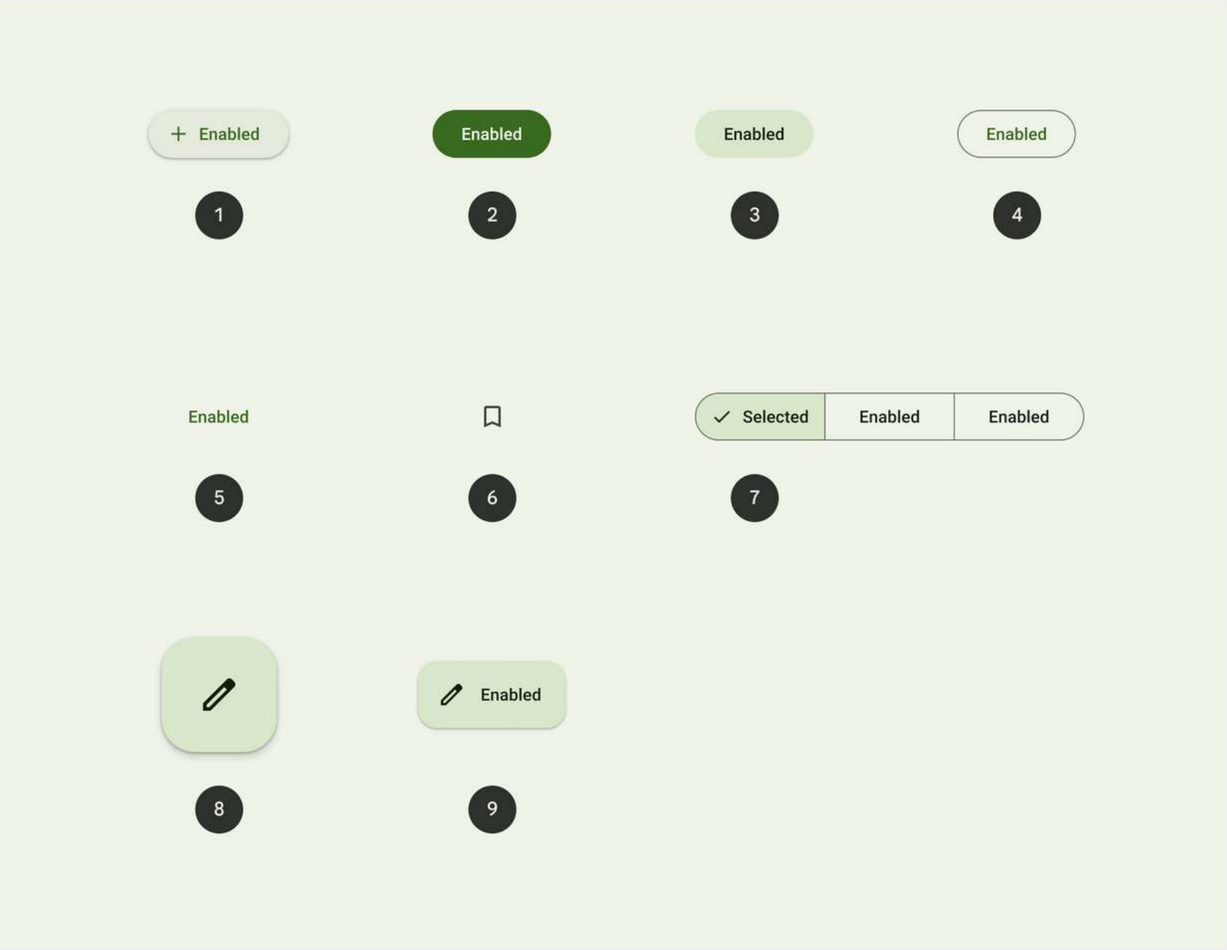
- **TextView** - This control is used to display text to the user.
- **EditText** is a predefined subclass of TextView that includes rich editing capabilities.
- The **AutoCompleteTextView** is a view that is similar to EditText, except that it shows a list of completion suggestions automatically while the user is typing.
- **Button** - A push-button that can be pressed, or clicked, by the user to perform an action.
- An **ImageButton** is an AbsoluteLayout which enables you to specify the exact location of its children. This shows a button with an image (instead of text) that can be pressed or clicked by the user.

UI components

- **CheckBox** - An on/off switch that can be toggled by the user. You should use check box when presenting users with a group of selectable options that are not mutually exclusive.
- **ToggleButton** - An on/off button with a light indicator.
- The **RadioButton** has two states: either checked or unchecked.
- A **RadioGroup** is used to group together one or more RadioButtons.
- The **ProgressBar** view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background.
- **Spinner** - drop-down list that allows users to select one value from a set.
- The **TimePicker** view enables users to select a time of the day, in either 24-hour mode or AM/PM mode.
- The **DatePicker** view enables users to select a date of the day.

Buttons

1. Elevated button
2. Filled button
3. Filled tonal button
4. Outlined button
5. Text button
6. Icon button
7. Segmented button
8. Floating action button (FAB)
9. Extended FAB

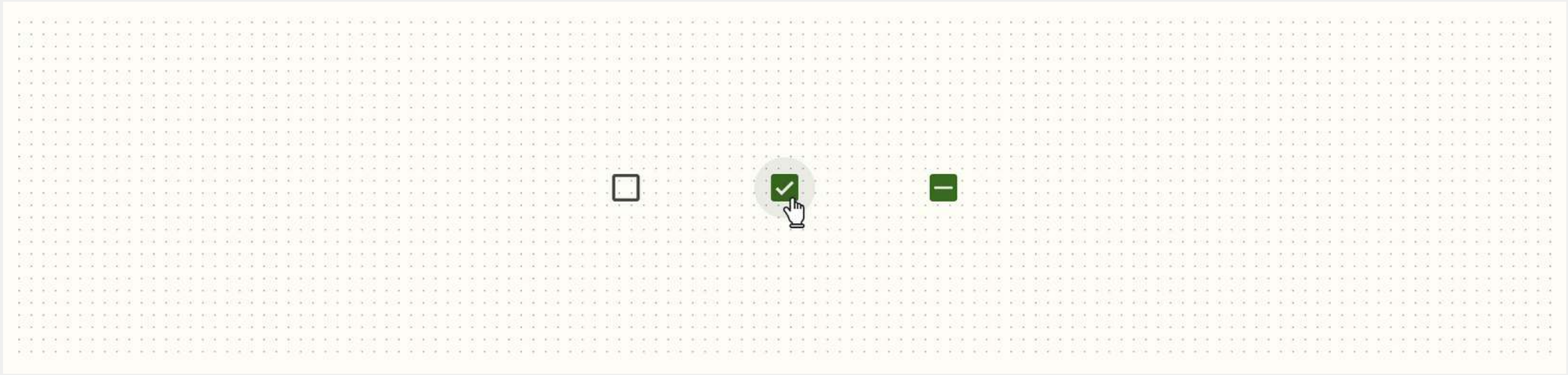


Text fields

1. Filled text field
2. Outlined text field

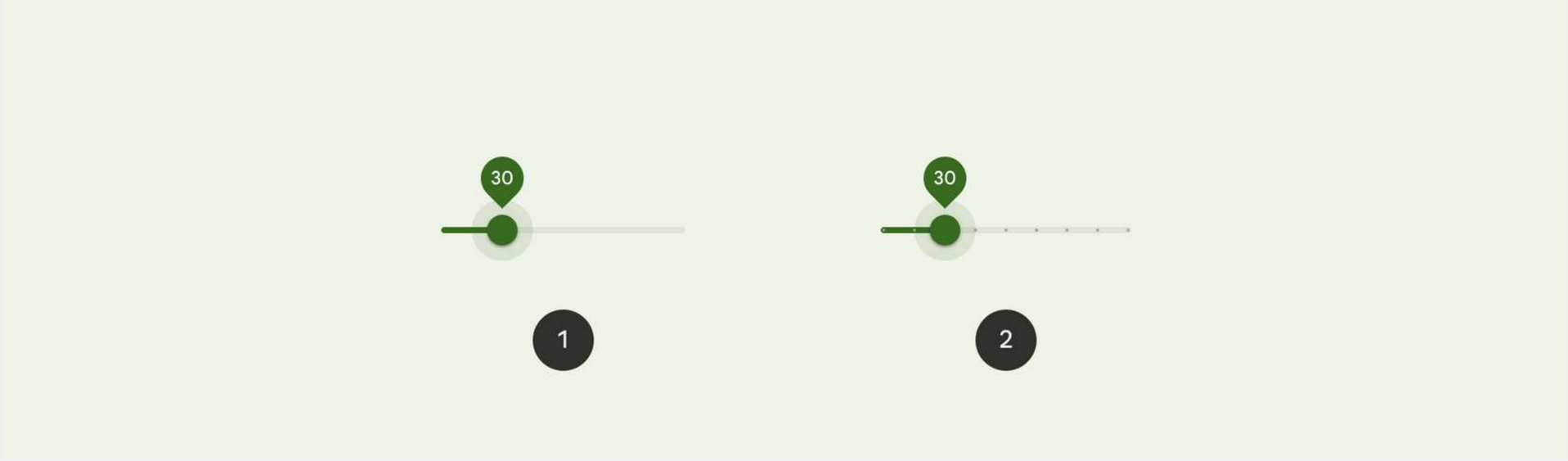
Checkboxes

Checkboxes shown in unselected, selected (hover), and indeterminate states.



Sliders

1. Continuous slider
2. Discrete slider



Constraint layout

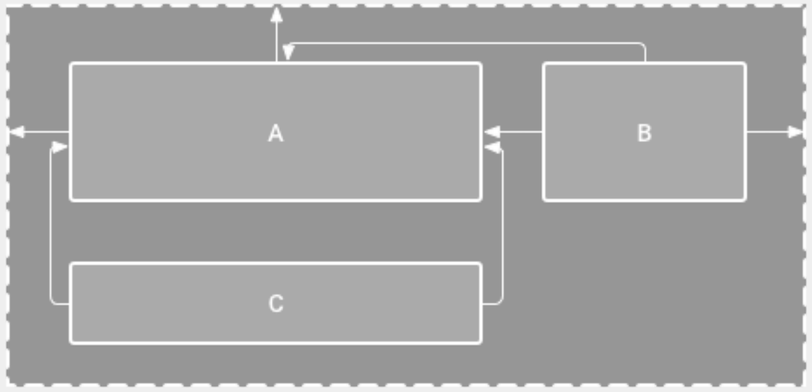
ConstraintLayout allows you to create large and complex layouts with a flat view hierarchy (no nested view groups). It's similar to **RelativeLayout** in that all views are laid out according to relationships between sibling views and the parent layout, but it's more flexible than RelativeLayout and easier to use with Android Studio's Layout Editor.
All the power of ConstraintLayout is available directly from the Layout Editor's visual tools, because the layout API and the Layout Editor were specially built for each other. So you can build your layout with ConstraintLayout entirely by drag-and-dropping instead of editing the XML.

Constraint layout

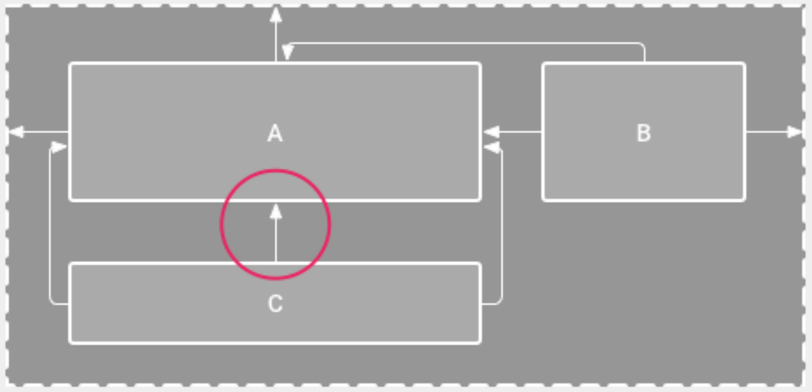
To define a view's position in ConstraintLayout, you must add **at least one horizontal and one vertical constraint** for the view. Each constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline. Each constraint defines the view's position along either the vertical or horizontal axis; so each view must have a minimum of one constraint for each axis, but often more are necessary.
When you drop a view into the Layout Editor, it stays where you leave it even if it has no constraints. However, this is only to make editing easier; if a view has no constraints when you run your layout on a device, it is drawn at position [0,0] (the top-left corner).

Constraint layout

In figure left, the layout looks good in the editor, but there's no vertical constraint on view **C**. When this layout draws on a device, view **C** horizontally aligns with the left and right edges of view **A**, but appears at the top of the screen because it has no vertical constraint.



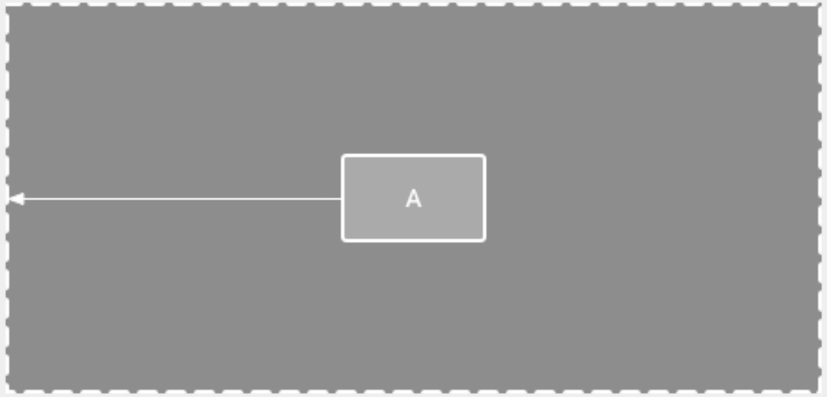
The editor shows view C below A, but it has no vertical constraint



View C is now vertically constrained below view A

Parent position

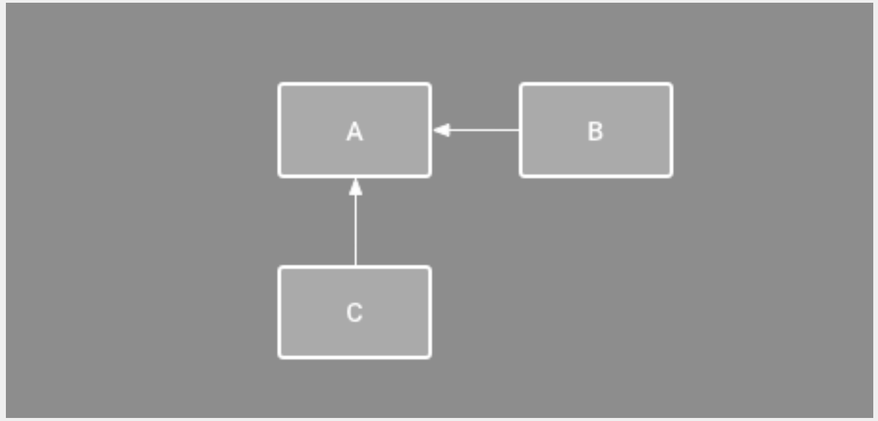
Constrain the side of a view to the corresponding edge of the layout.
In figure below, the left side of the view is connected to the left edge of the parent layout. You can define the distance from the edge with margin.



A horizontal constraint to the parent

Order position

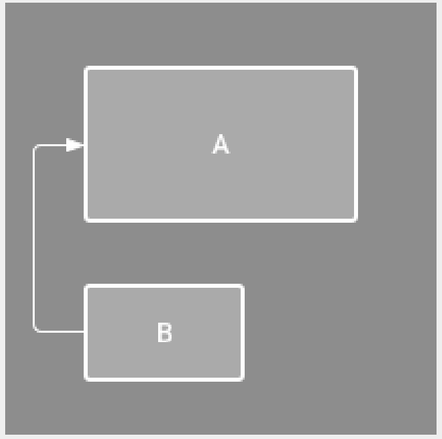
Define the order of appearance for two views, either vertically or horizontally.
In figure below, **B** is constrained to always be to the right of **A**, and **C** is constrained below **A**. However, these constraints do not imply alignment, so **B** can still move up and down.



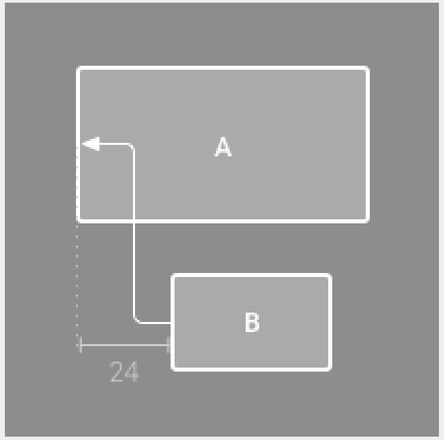
A horizontal and vertical constraint

Alignment

Align the edge of a view to the same edge of another view.
In figure left, the left side of **B** is aligned to the left side of **A**. If you want to align the view centers, create a constraint on both sides.
You can offset the alignment by dragging the view inward from the constraint. For example, figure right shows **B** with a **24dp** offset alignment. The offset is defined by the constrained view's margin.



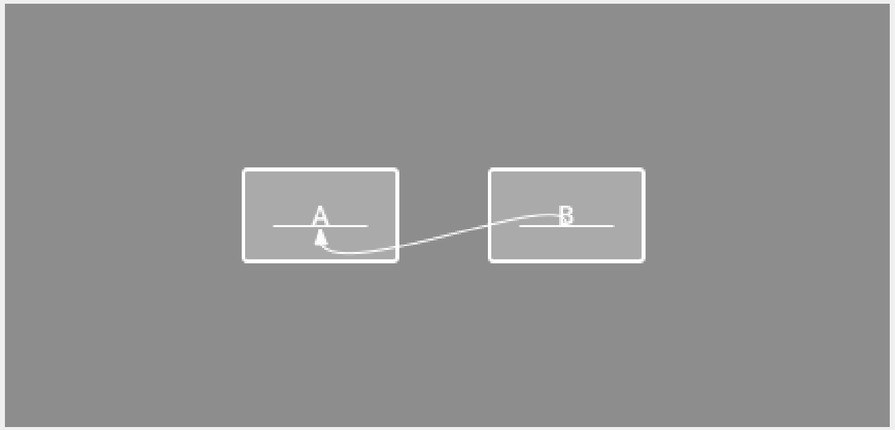
A horizontal alignment constraint



An offset horizontal alignment constraint

Baseline alignment

Align the text baseline of a view to the text baseline of another view.
In figure below, the first line of **B** is aligned with the text in **A**.
To create a baseline constraint, right-click the text view you want to constrain and then click **Show Baseline**. Then click on the text baseline and drag the line to another baseline.



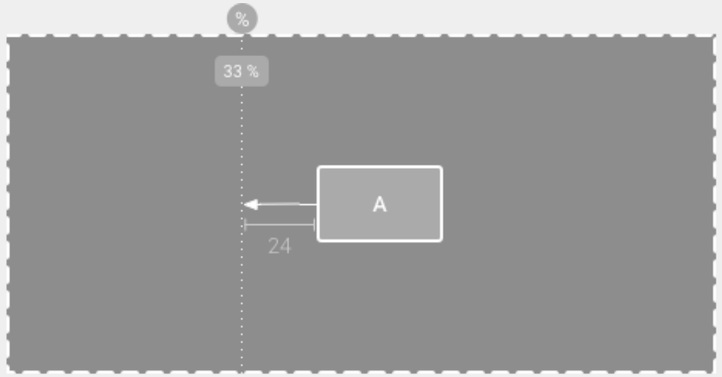
A baseline alignment constraint

Constrain to a guideline

You can add a vertical or horizontal guideline to which you can constrain views, and the guideline will be invisible to app users. You can position the guideline within the layout based on either dp units or percent, relative to the layout's edge.

To create a guideline, click **Guidelines** in the toolbar, and then click either **Add Vertical Guideline** or **Add Horizontal Guideline**.

Drag the dotted line to reposition it and click the circle at the edge of the guideline to toggle the measurement mode.



A view constrained to a guideline

References

- **Layouts** - https://developer.android.com/develop/ui/views/layout/declaring-layout
- **UI components** - www.tutorialspoint.com/android/android_user_interface_controls.htm
- **UI components design** - https://m3.material.io/components
- **Constraint layout** - https://developer.android.com/develop/ui/views/layout/constraint-layout

End of the session