Laboratorio di Applicazioni Mobili
Bachelor in Computer Science &
Computer Science for Management

University of Bologna

Android Views UI

Federico Montori
federico.montori2@unibo.it

Table of Contents

Android Views

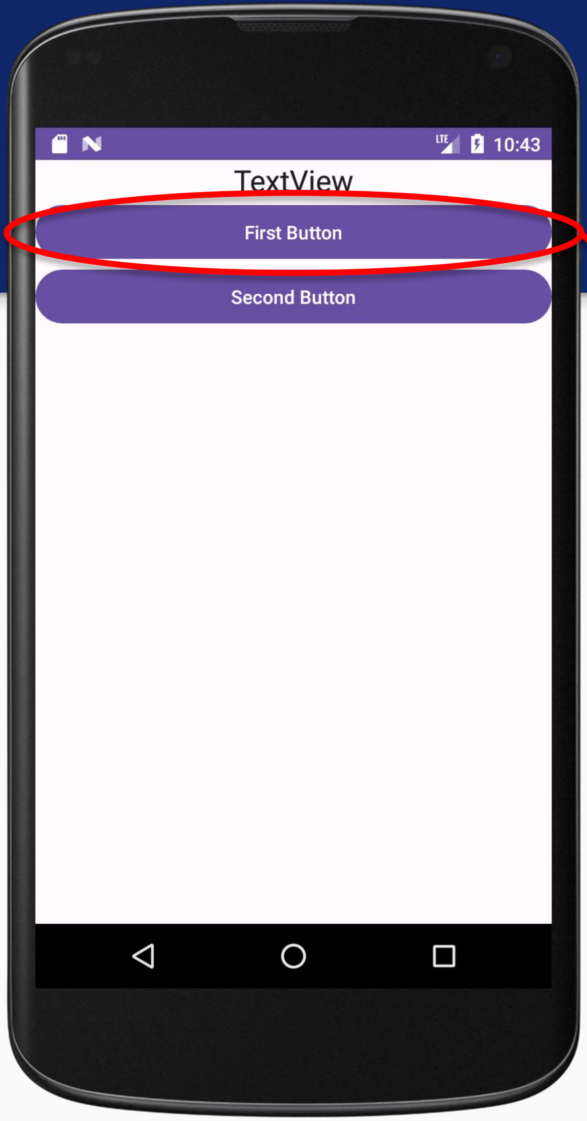
Android Views is the common and standard paradigm for building a responsive UI in Android.

It is the one originated since the first version of Android (2008).

Uses the concept of **View** = any self-contained object on the screen (including containers of other Views).

Since 2021 it is possible to use a new UI toolkit called **Jetpack Compose**, only for Kotlin, which works in a completely different way. We will see that too.
https://developer.android.com/jetpack/compose

Views



- Rectangular area of the screen
- Responsible for **drawing**
- Responsible for **event handling**

Examples of Views

- GoogleMap
- WebView
- **Widgets** →topic of the day!
- **Layouts** →topic of the day!

Views → basic building blocks for user interface components

Declarative Views

Views need to be declared in the **XML layout** file (declarative mode)

```
<TextView
     android:id = "@+id/myTextView"
     android:layout_width = "match_parent"
     android:layout_height = "wrap_content"
     android:text = "Hello World"
     android:textAlignment = "center"
/>
<!-- This is in res/layout/activity_main.xml -->
```

Declarative Views

Declarative mode: Views can be declared in XML and accessed in Java/Kotlin through **findViewById**

XML

```
<TextView
    android:id = "@+id/myTextView" />
```

JAVA

```
public TextView textView;
textView = (TextView)findViewById(R.id.myTextView);
```

CAST REQUIRED UNTIL API 26

KOTLIN

```
lateinit var textView : TextView
textView = findViewById(R.id.myTextView)
```

Programmatic Views

Programmatic mode: Views can be directly created in Java/Kotlin (giving it a context), however they must also be given all their visual properties in the code, not promoting the separation of concerns. This is not recommended.

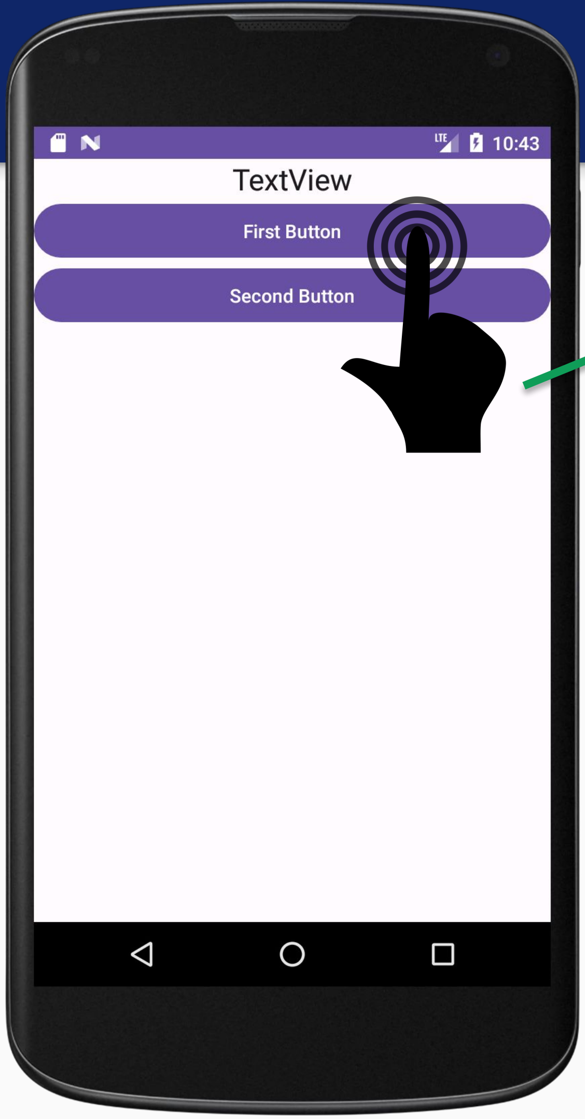
```
public TextView textView;
textView = new TextView(this);
```

```
lateinit var textView : TextView
textView = TextView(this);
```

Handling Events

```
<Button
    android:id=
        "@+id/button1"
    android:text=
        "First Button"
/>
```

```
lateinit var button
    : Button
button = findViewById(
    R.id.button1
)
```



ONCLICK event

Java/Kotlin code that manages the **onClick** event

You can also fire an event from the code:

```
button.performClick()
```

Handling Events

Views are interactive components: upon certain actions, an appropriate event will be fired (click, long click, focus, items selected, items checked, drag, …)

PROBLEM: How to **handle** these events?

1. Directly from **XML**
2. Through **Event Handlers** (general)
3. Through **Event Listeners** (general, recommended)

Handling Events

For a limited set of components, it is possible to manage the events through **callbacks**, directly indicated in the XML.

XML

```
<Button
    android:id = "@+id/button1"
    android:text= "First Button"
    android:onClick="doSomething"
/>
```

```
void doSomething (View v)
    { … }
```

JAVA

```
fun doSomething (v : View)
    { … }
```

KOTLIN

Handling Events

Views are interactive components: upon certain actions, an appropriate event will be fired (click, long click, focus, items selected, items checked, drag, …)

PROBLEM: How to **handle** these events?

1. Directly from **XML**
2. Through **Event Handlers** (general)
3. Through **Event Listeners** (general, recommended)

Handling Events

- Each View contains several methods that are called when an event occurs:
    - e.g. **onTouchEvent**() when the View is clicked
- In order to intercept it we should extend the View class and override the call.
- This is impractical… much better to have a separate class that handles all the hassle.

$$\begin{tikzcd}\boxed{\text{Button}} \arrow[r] & \boxed{\text{onTouchEvent()}}\end{tikzcd}$$

Handling Events

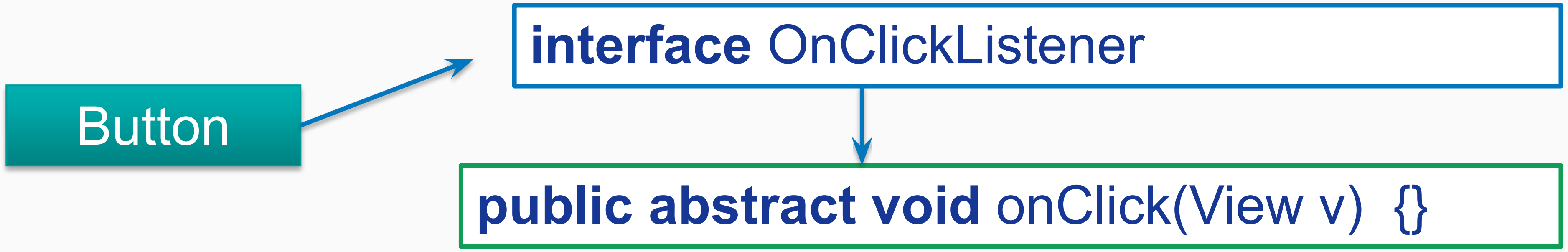
Views are interactive components: upon certain actions, an appropriate event will be fired (click, long click, focus, items selected, items checked, drag, …)

PROBLEM: How to **handle** these events?

1. Directly from **XML**
2. Through **Event Handlers** (general)
3. Through **Event Listeners** (general, recommended)

Event Listeners

- Each View can delegate the reaction to one event to an object that implements the dedicated **listener** interface.
  - Each listener is a Single Abstract Method (SAM) interface
  - Each listener handles a single **type of events**
  - Each listener contains a single **callback** method


Button
interface OnClickListener
public abstract void onClick(View v) {}

e.g. assign the **OnClickListener** to the **View** through **setOnClickListener**

Event Listeners

Java Example: make the Activity implement it

```
Button button;
class MainActivity extends AppCompatActivity implements OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        button = findViewById(R.id.button1);
        button.setOnClickListener(this);
    }
    @Override
    void onClick(View v) {   /* Behavior */   }
}
```

Event Listeners

Kotlin Example: make the Activity implement it

```
lateinit var button : Button

class MainActivity : AppCompatActivity(), OnClickListener {
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        button = findViewById(R.id.button1)
        button.setOnClickListener(this)
  }
    override fun onClick(v: View?) {   /* Behavior */   }
}
```

Event Listeners

Java Example: make an **anonymous Object** (more common)

```
button.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View view) {   /* Behavior */   }
    });
```

Kotlin Example: make an **anonymous Object** (more common)

```
button.setOnClickListener(object: OnClickListener {
        override fun onClick(v : View?) {   /* Behavior */   }
    })
```

Event Listeners

Java 8 Example: anonymous Object with **Lambda notation** (most common)

https://www.w3schools.com/java/java_lambda.asp#:~:text=Lambda%20Expressions%20were%20added%20in,the%20body%20of%20a%20method

```
button.setOnClickListener(
    v -> {   /* Behavior */   }
    );
```

Kotlin Example: anonymous Object with **Lambda notation** (most common)

```
button.setOnClickListener{
        /* Behavior */
    }
```

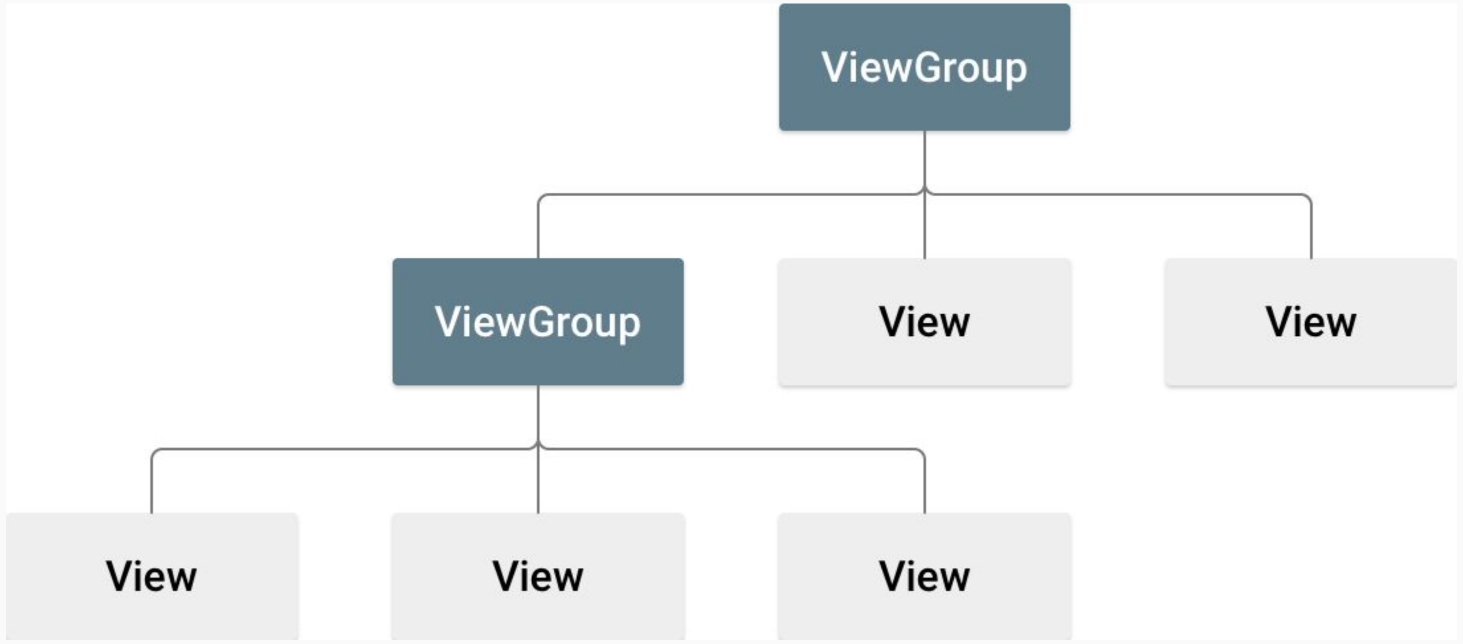
Event Listeners

Some Event Listeners:

- interface OnClickListener
  - abstract method: onClick()
- interface OnLongClickListener
  - abstract method: onLongClick()
- interface OnFocusChangeListener
  - abstract method: onFocusChange()
- interface OnKeyListener
  - abstract method: onKey()
- interface OnCheckedChangeListener
  - abstract method: onCheckedChanged()
- interface OnItemSelectedListener
  - abstract method: onItemSelected()
- interface OnTouchListener
  - abstract method: onTouch()
- interface OnCreateContextMenuListener
  - abstract method: onCreateContextMenu()

View Hierarchy

ViewGroup objects are invisible containers that define a layout structure for the Views declared in it.



NB. ViewGroup is a (subclass of) View

Layouts

A **Layout** must extend a ViewGroup (i.e. a Layout IS a ViewGroup)

Every View in a Layout needs to specify:

- android:layout_height
- android:layout_width
- A dimension or one of match_parent or wrap_content

Layouts

When building your app you first declare your layout(s) in XML in the folder “res/layouts”:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
     "http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <TextView
    android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="TextView"
    android:textAlignment="center" />
```

```
  <Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="First Button" />
  <Button
    android:id="@+id/button2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Second Button" />
</LinearLayout>
```

Note that you can still declare the layout programmatically...

Example of the declaration of a LinearLayout.

Layouts

Your layout is then compiled into a View resource that has to be loaded by the Activity making use of it.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
}
```

Each View can have an ID

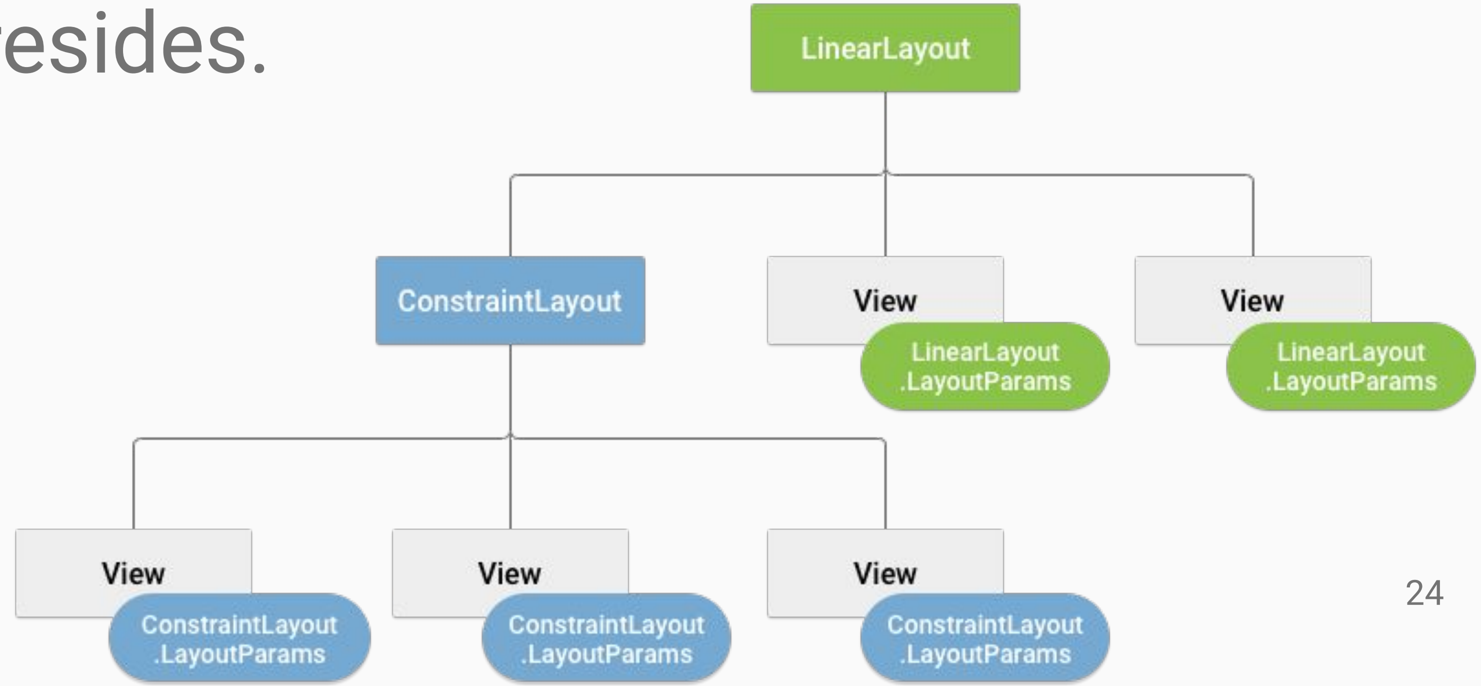
```
android:id="@+id/button1"
```

@ means: “parse and expand the rest of the string as an id resource.
+ means: “this is going to be added as a new id in R.java” (we’ll see what it is)

Layouts

XML layout attributes named **layout_something** define layout parameters for the View that are appropriate for the **ViewGroup** in which it resides.

Each parent **Layout** specifies **LayoutParams** that each children **View** must implement.



LinearLayout
ConstraintLayout
View
LinearLayout .LayoutParams
View
LinearLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams
View
ConstraintLayout .LayoutParams

Layouts

E.g. each Layout needs the children Views to implement **layout_width** and **layout_height**.

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

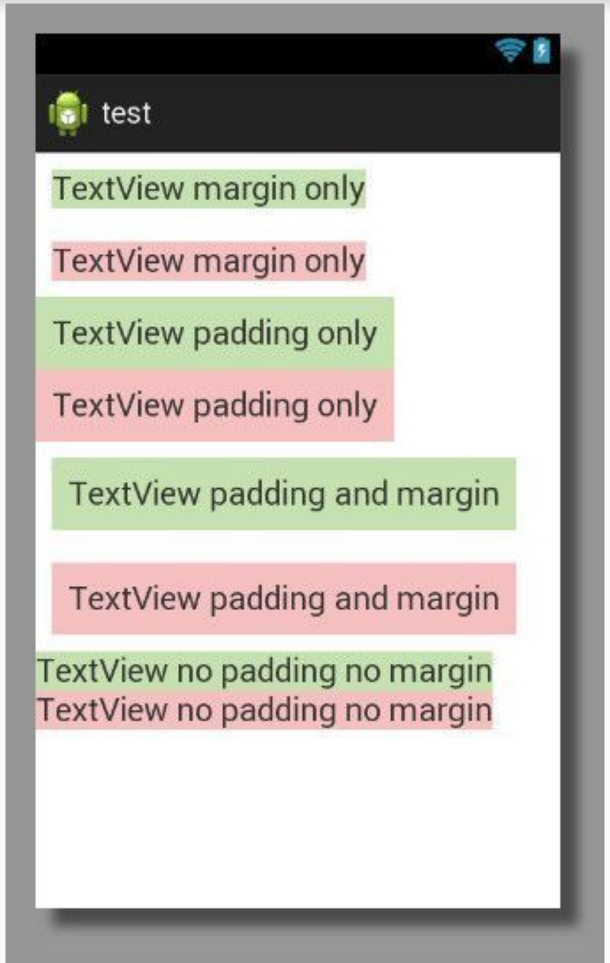
Typically
- match_parent
- wrap_content
- 0dp
- a custom value

0dp is a wildcard that means: “take up the available space” only in certain cases (constraint or weights). Dp means density independent pixel (we will explore this better).

Layouts

Android also supports Padding and Margin:

- Padding is a View property
  - **android:padding**
- Margin is a layout property
  - **android:layout_margin**



TextView margin only
TextView margin only
TextView padding only
TextView padding only
TextView padding and margin
TextView padding and margin
TextView no padding no margin
TextView no padding no margin

Layouts

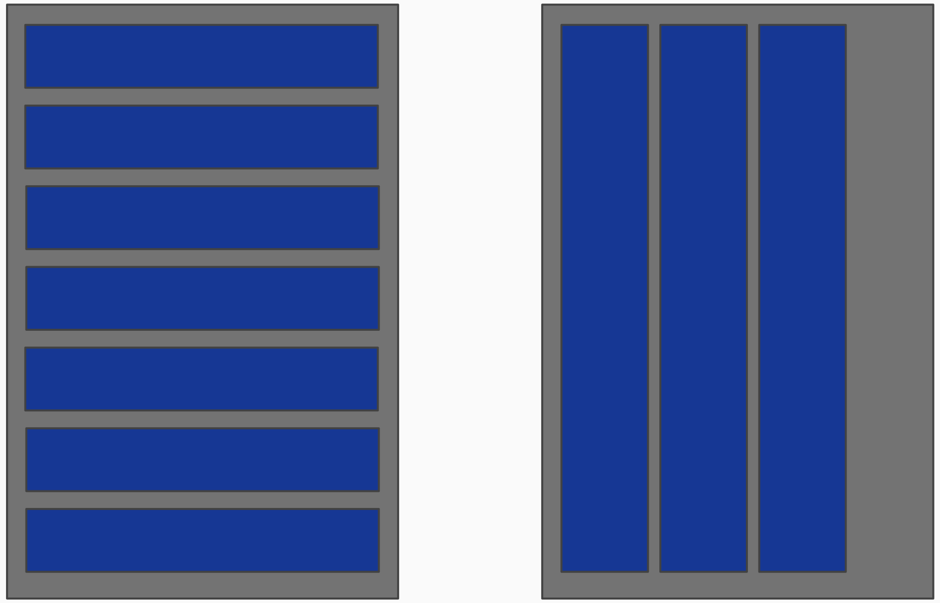
Here are the most common static layouts (that do not change at runtime) predefined in Android:

- LinearLayout ⭐
- RelativeLayout
- TableLayout
- FrameLayout
- ConstraintLayout ⭐

A layout can be declared inside another layout

LinearLayout

Organizes Views on a single row or column, depending on **android:layout_orientation**: one of VERTICAL or HORIZONTAL

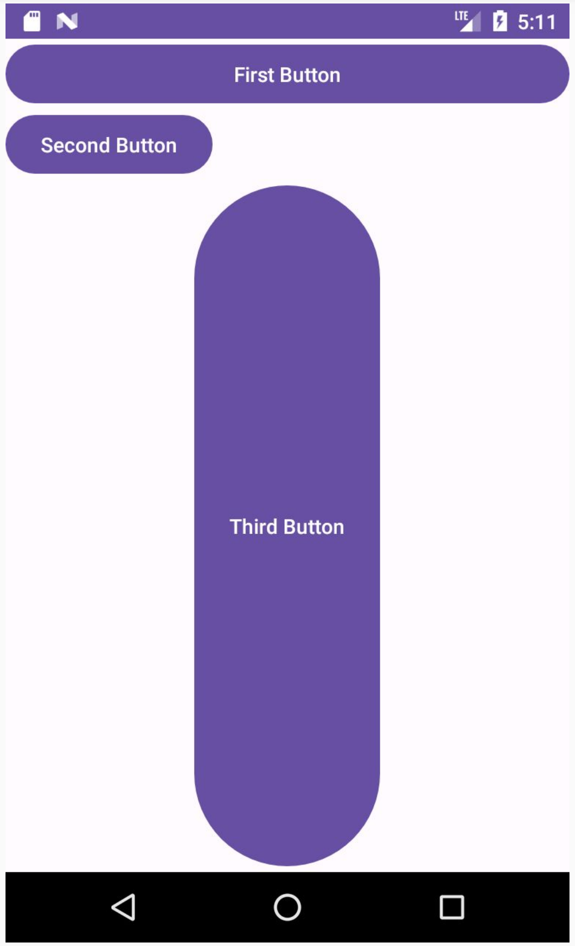


Has two other attributes:

- **layout_gravity**
- **layout_weight**

LinearLayout

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <Button
      android:id="@+id/button1"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="First Button" />
   <Button
      android:id="@+id/button2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Second Button" />
   <Button
      android:id="@+id/button3"
      android:layout_width="wrap_content"
      android:layout_height="match_parent"
      android:layout_gravity="center_horizontal"
      android:text="Third Button" />
</LinearLayout>
```

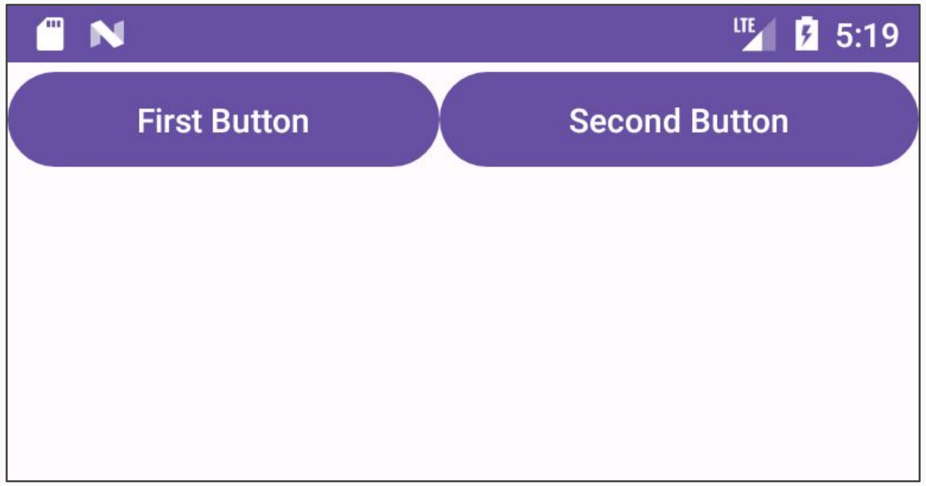


LinearLayout

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="horizontal"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <Button
      android:id="@+id/button1"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="First Button" />
   <Button
      android:id="@+id/button2"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:text="Second Button" />
</LinearLayout>
```

If one of the Views has a **weight**, then the Views will all take up the entire dimension.

Weights tell us how important that View is and go best with **0dp**

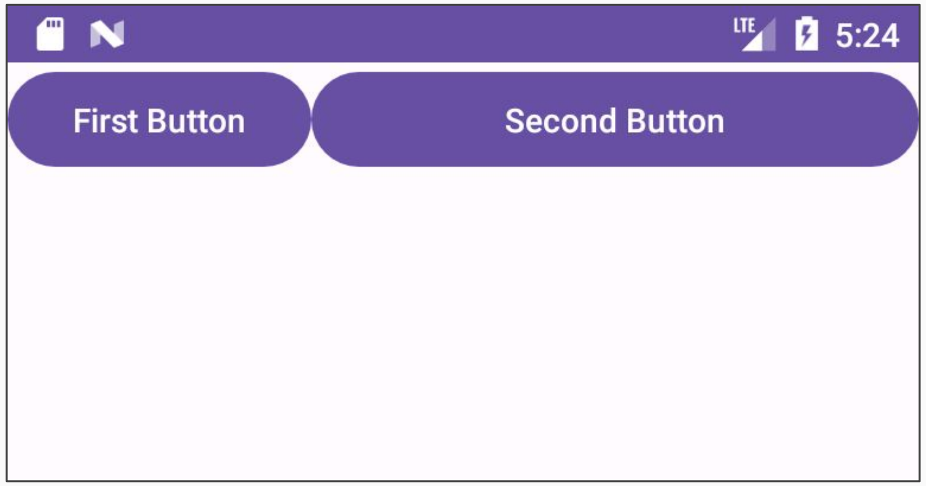


LinearLayout

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <Button
    android:id="@+id/button1"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="First Button" />
  <Button
    android:id="@+id/button2"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="2"
    android:text="Second Button" />
</LinearLayout>
```

If one of the Views has a **weight**, then the Views will all take up the entire dimension.

Weights tell us how important that View is and go best with **0dp**



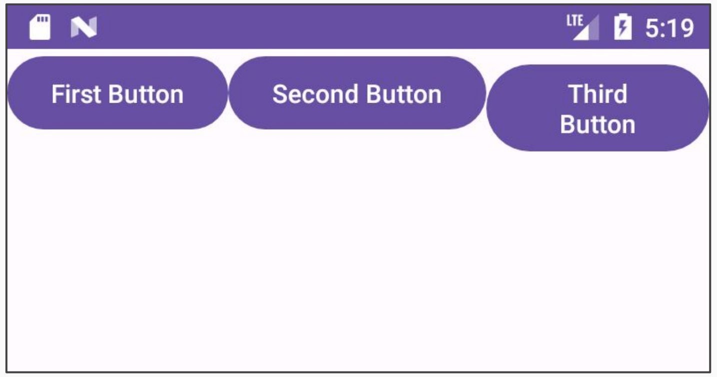
LinearLayout

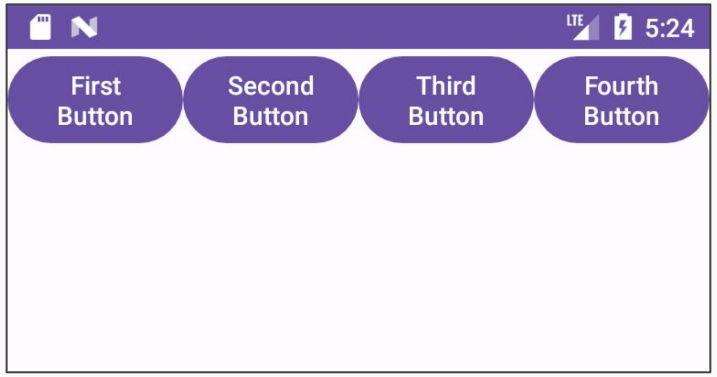
What if elements do not fit? Wrap the layout in a **<ScrollView>** or **<HorizontalScrollView>**

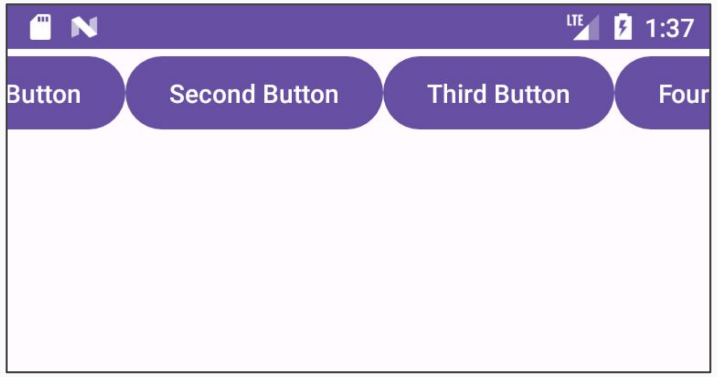
```
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

        … <!-- All buttons wrap content -->…

</LinearLayout>
```

```
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

        … <!-- All buttons weighted, 0dp -->…

</LinearLayout>
```

```
<HorizontalScrollView
        xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
        <LinearLayout … >
        … <!-- All buttons wrap content -->…
        </LinearLayout>
</HorizontalScrollView>
```







ConstraintLayout

Introduced since Android 2.3, it organizes Views according to **constraints**.

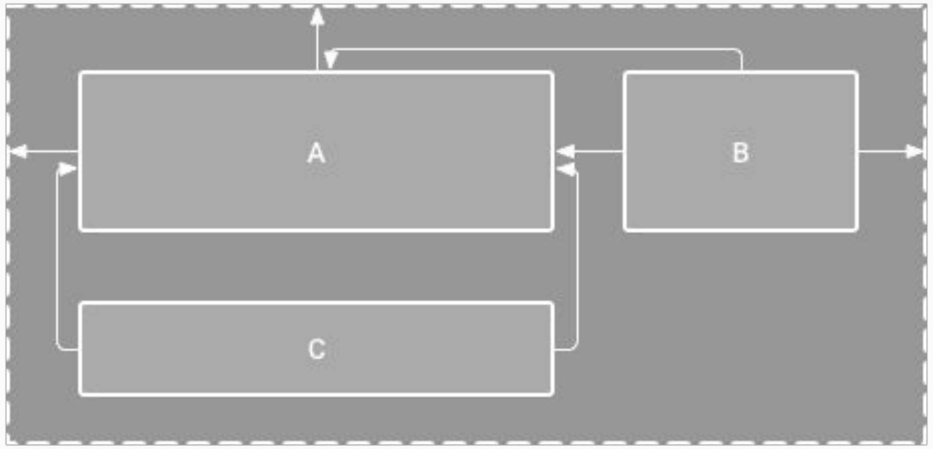
- Overarching idea: define constraints (top/bottom/left/right) for each view
- Each constraint has to be defined to another (previously declared) view, another layout or an invisible guideline.
- It is a flat View hierarchy

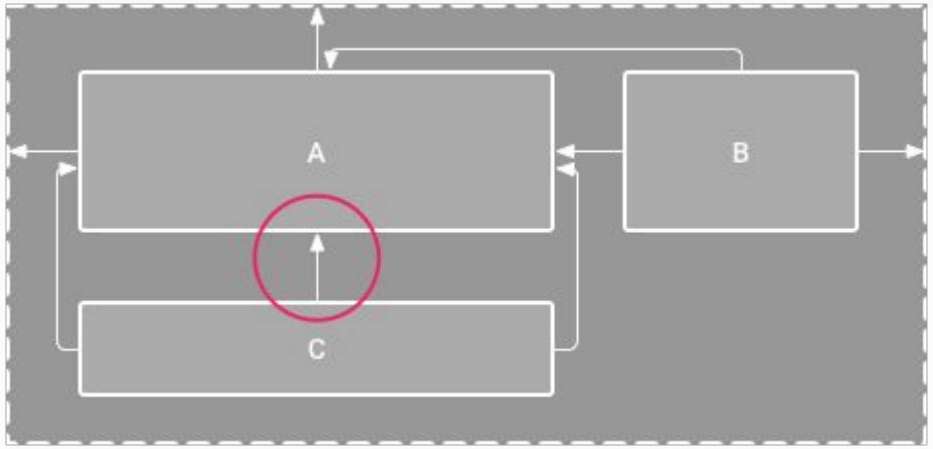
You may have noticed that it is the default one...

ConstraintLayout

- Each view needs at least one constraint per plane.
  - (plane = vertical | horizontal)
- Constraints can be defined only between anchor points sharing the same plane.
- Each handle can define one constraint.
- Multiple handles can define a constraint to a single anchor point.
- Adding 2 opposite constraints places the view in the middle and can adjust the ratio by setting the **bias**.

ConstraintLayout

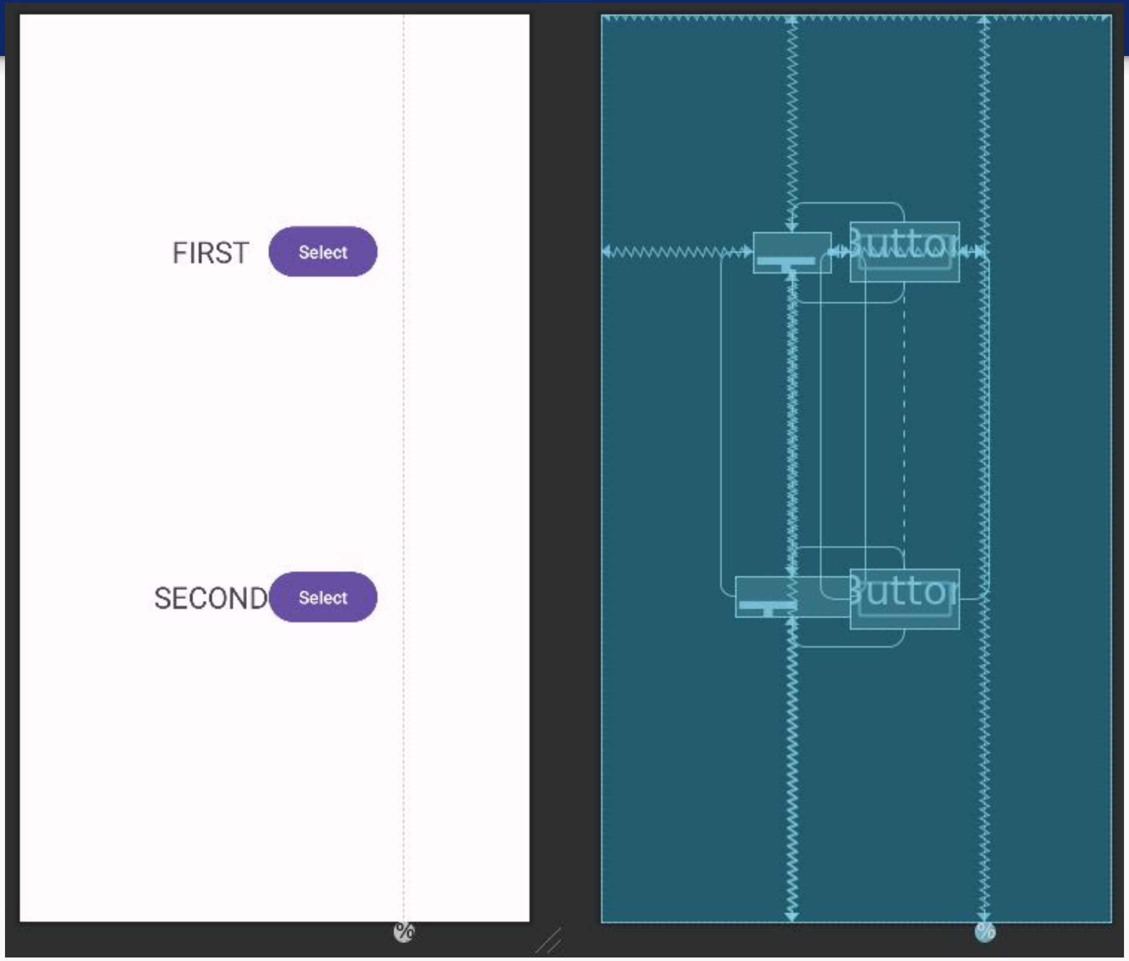




Both layouts will show no error, however the left one has no top constraint on C, which will then be placed at the top.

Two opposing constraints not reporting any distance will place the element right in the middle.

ConstraintLayout



ConstraintLayout has a complex and verbose syntax. In the **layout editor** of Android Studio you'll see on the right the constraints, and on the left a preview and can edit them by drag-and-drop.

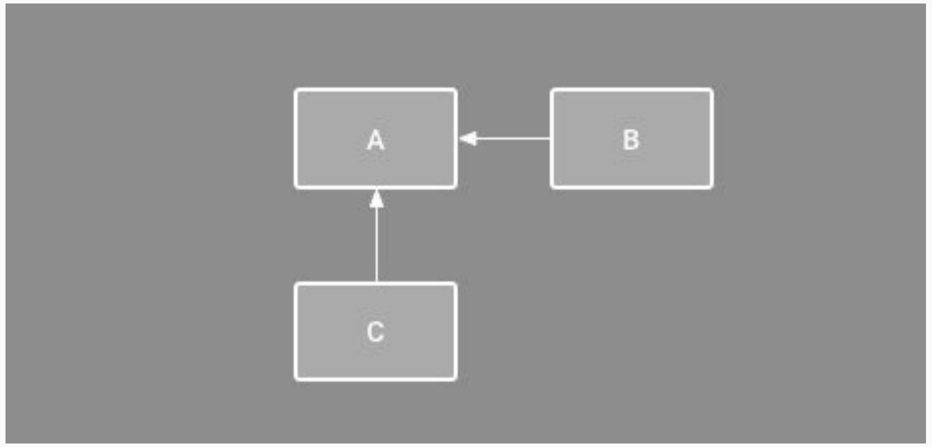
Here, a size of **0dp** means always “match constraint”, more or less like “take all the available space”

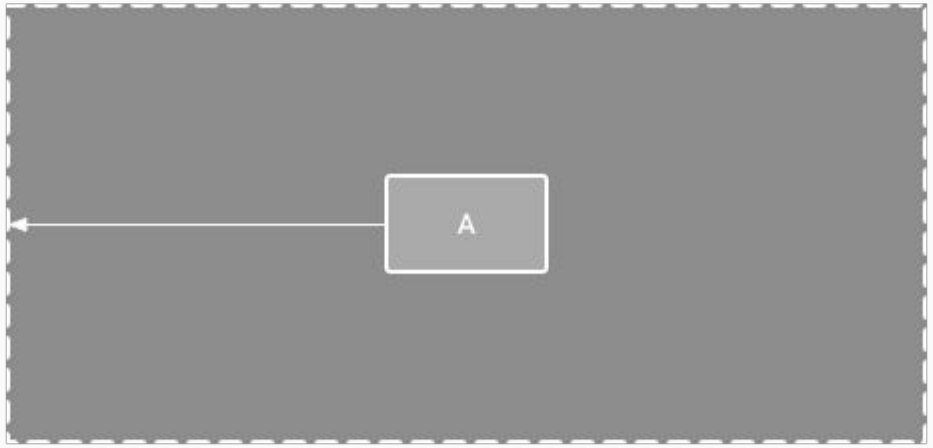
ConstraintLayout

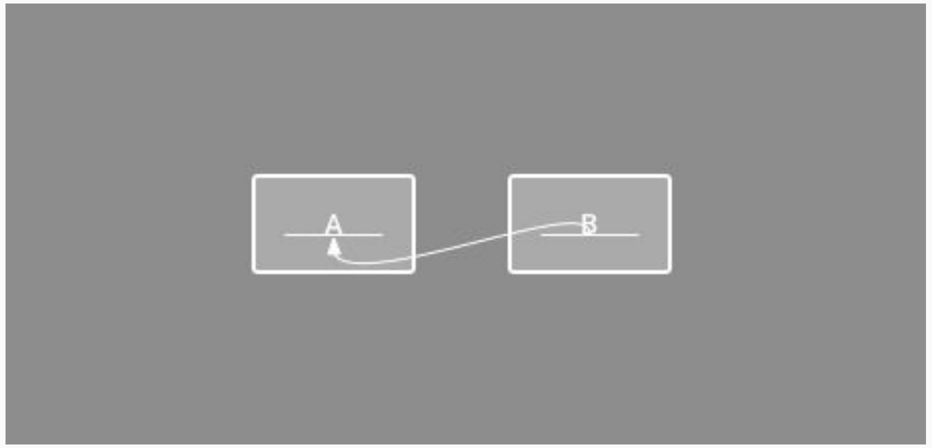
```
<androidx.constraintlayout.widget.ConstraintLayout
        xmlns:android=
                "http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
  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

  <androidx.constraintlayout.widget.Guideline
    android:id="@+id/guideline"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintGuide_percent="0.75" />

  <TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="FIRST"
    android:textSize="24sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/guideline"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.25" />
<TextView
   android:id="@+id/textView2"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="SECOND"
   android:textSize="24sp"
   app:layout_constraintBottom_toBottomOf=
       "parent"
   app:layout_constraintEnd_toEndOf=
       "@+id/textView"
   app:layout_constraintStart_toStartOf=
       "@+id/textView"
   app:layout_constraintTop_toBottomOf=
       "@+id/textView" />
<Button
   android:id="@+id/button"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Select"
   app:layout_constraintBottom_toBottomOf=
       "@+id/textView"
   app:layout_constraintEnd_toStartOf=
       "@+id/guideline"
   app:layout_constraintStart_toEndOf=
       "@+id/textView"
    app:layout_constraintTop_toTopOf=
       "@+id/textView" />
  <Button
    android:id="@+id/button5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Select"
    app:layout_constraintBottom_toBottomOf=
       "@+id/textView2"
    app:layout_constraintEnd_toEndOf=
       "@+id/button"
    app:layout_constraintStart_toStartOf=
       "@+id/button"
    app:layout_constraintTop_toTopOf=
       "@+id/textView2" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

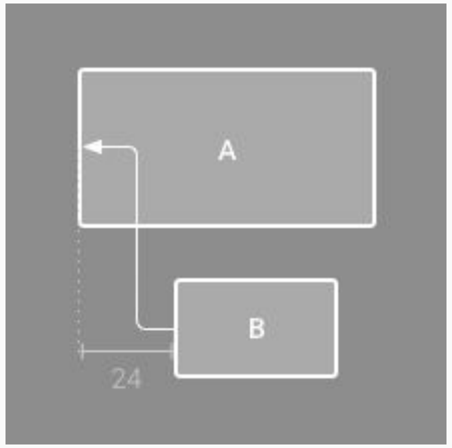
This is the code of the previous screen…

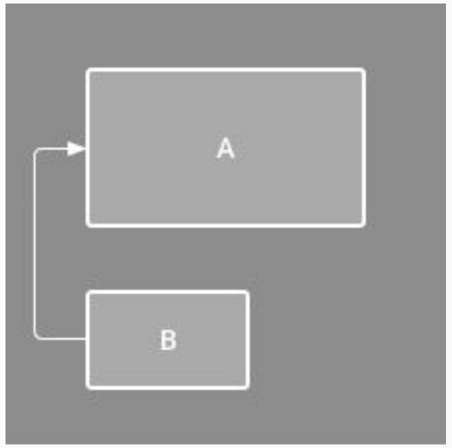
ConstraintLayout



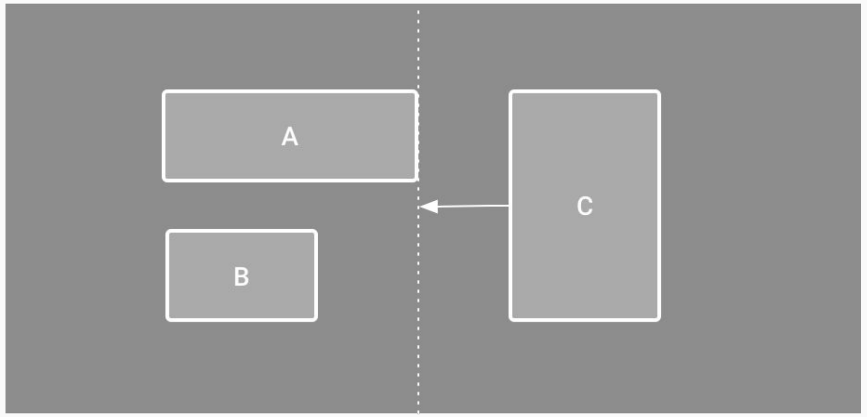


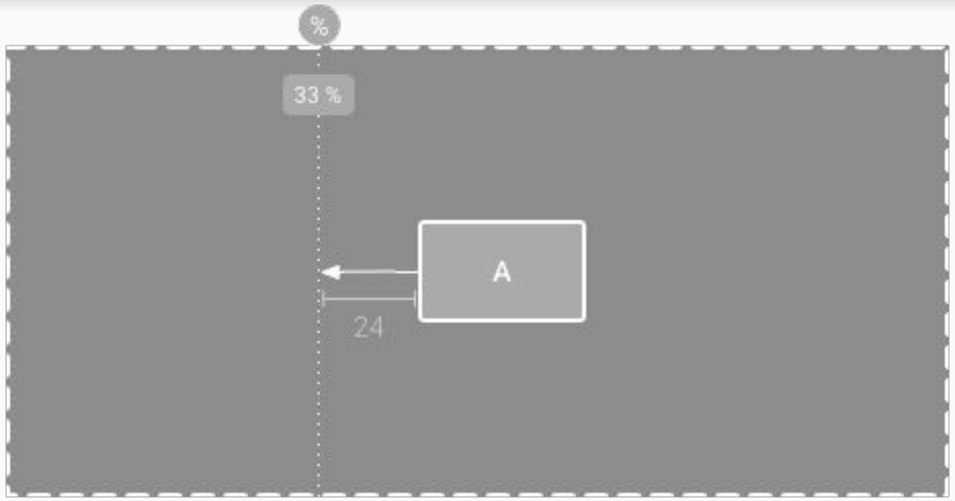


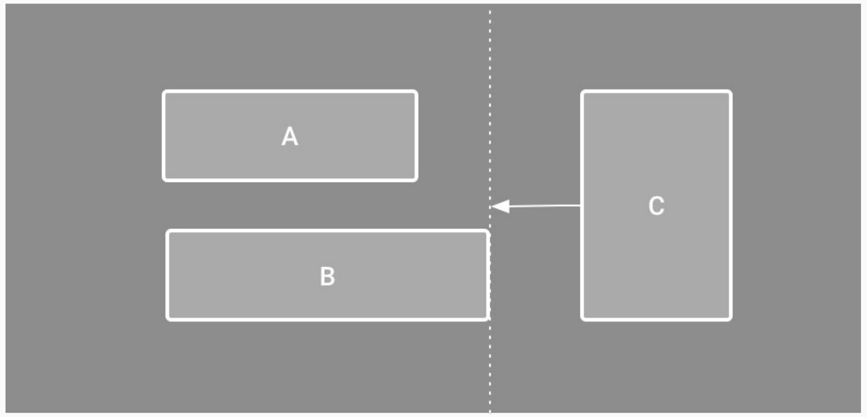


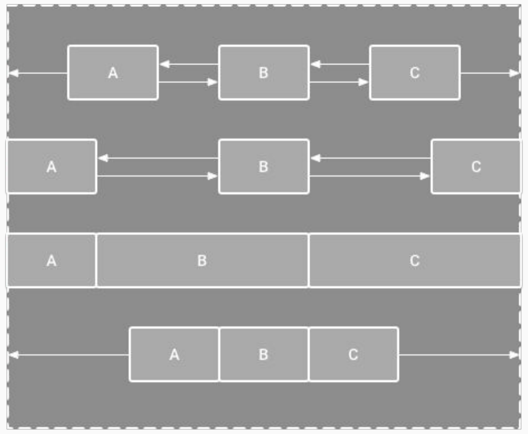


ConstraintLayout

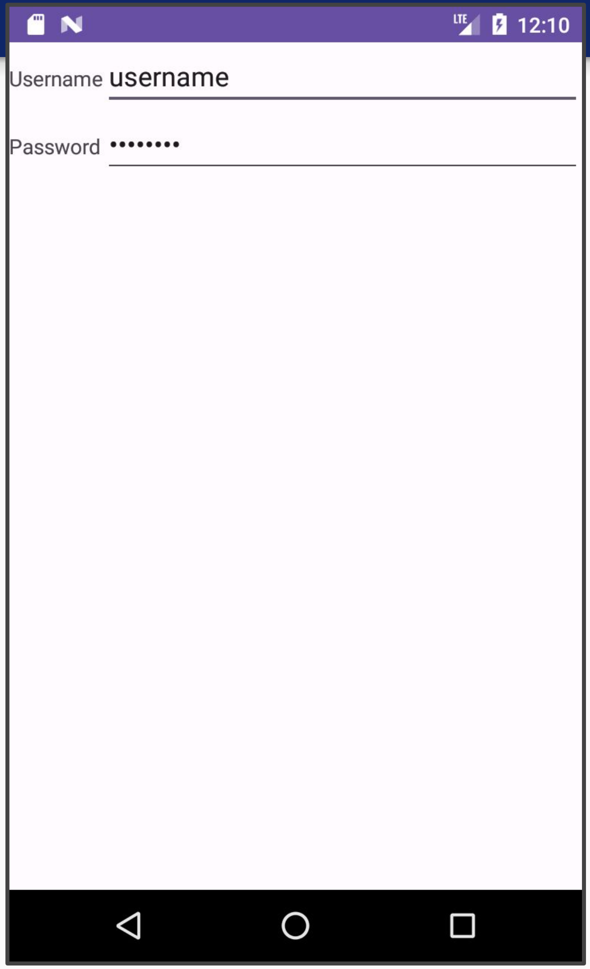






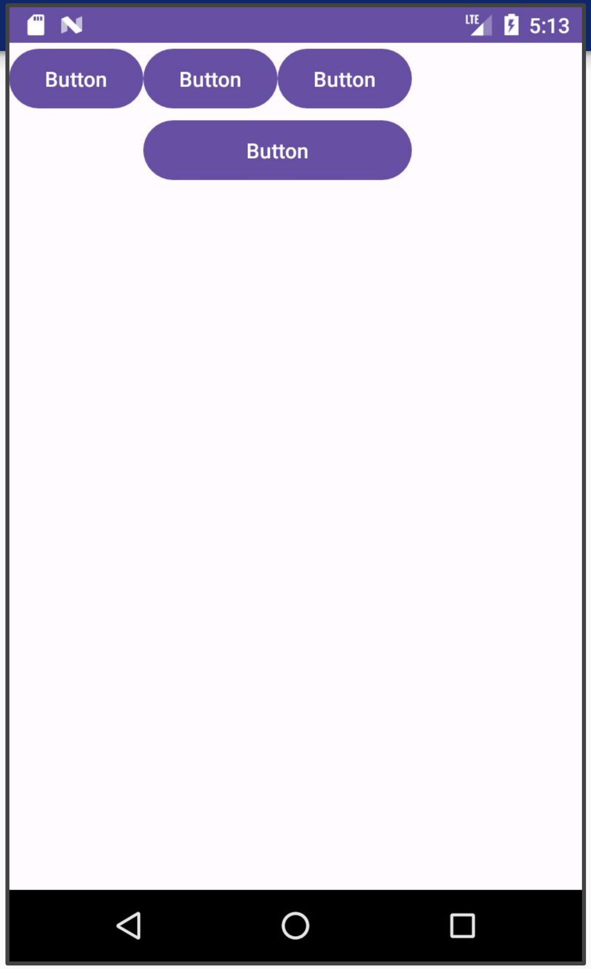


Other layouts: RelativeLayout



RelativeLayout displays child views in relative positions. The position of each view can be specified as relative to sibling elements (such as to the left-of or below another view) or in positions relative to the parent RelativeLayout area (such as aligned to the bottom, left or center).

Other layouts: TableLayout



TableLayout positions its children into rows and columns. TableLayout containers do not display border lines for their rows, columns, or cells. The table will have as many columns as the row with the most cells. A table can leave cells empty. Cells can span multiple columns, as they can in HTML.

Other layouts

FrameLayout

Adds an attribute, android:visibility

Blocks out portion of the screen to suit (typically) only one object.

Size equal to the size of its largest (non GONE) child.

AbsoluteLayout

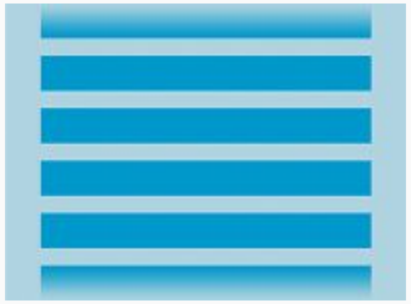
Deprecated

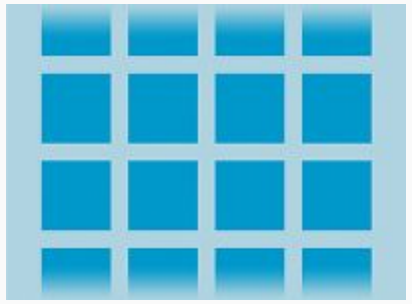
Specify position with x and y

Dynamic layouts

Sometimes the layout needs to be populated at runtime with Views (all the same type of View).

e.g. **ListView**, **GridView…**





These layouts subclass **AdapterView**: they use an **Adapter** to retrieve data from another source and map it into the elements of the AdapterView.

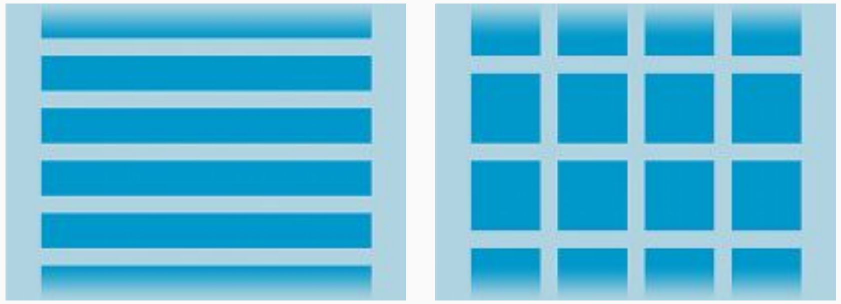
Dynamic layouts

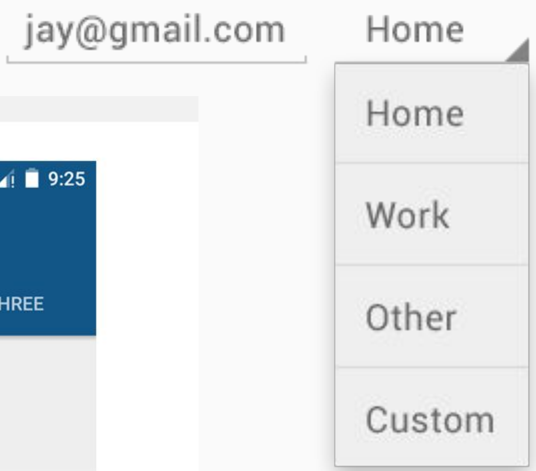
AdapterView: A ViewGroup subclass

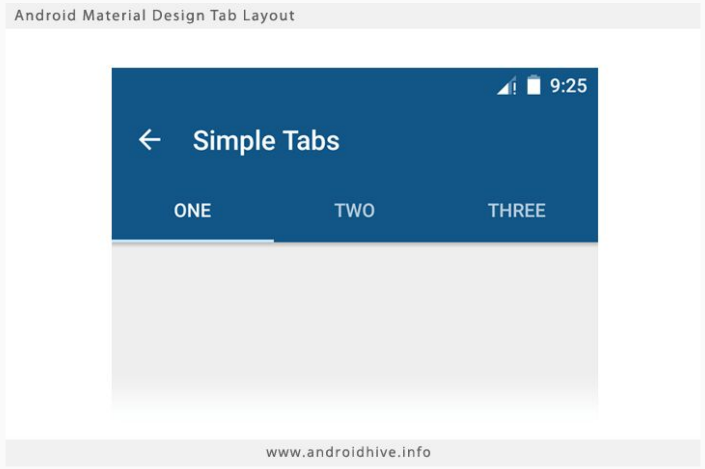
Its subchildren are determined by an Adapter

Some subclasses:

- **ListView**
- **GridView**
- **Spinner**
  - **(selection of multiple items)**
- **Gallery**
  - **(images)**
- **ExpandableListView**
- **TabLayout**







ListView

Adapter: used to visualize dynamic data (e.g. **ArrayAdapter**)

Some methods:

- **isEmpty()**
- **getItem(int position)**
- **getCount()**
- **getView()**

Pair it with a data structure where data is saved.

```
// Create a list adapter for a string list

String[] data = {“First”, “Second”, “Third”};
ArrayAdapter<String> adapter =
    new ArrayAdapter<String>(
        this,
        android.R.layout.simple_list_item_1,
        data
    );
ListView listView =
    findViewById(R.id.listView);
listView.setAdapter(adapter);
```

ListView

The Adapter takes in input:

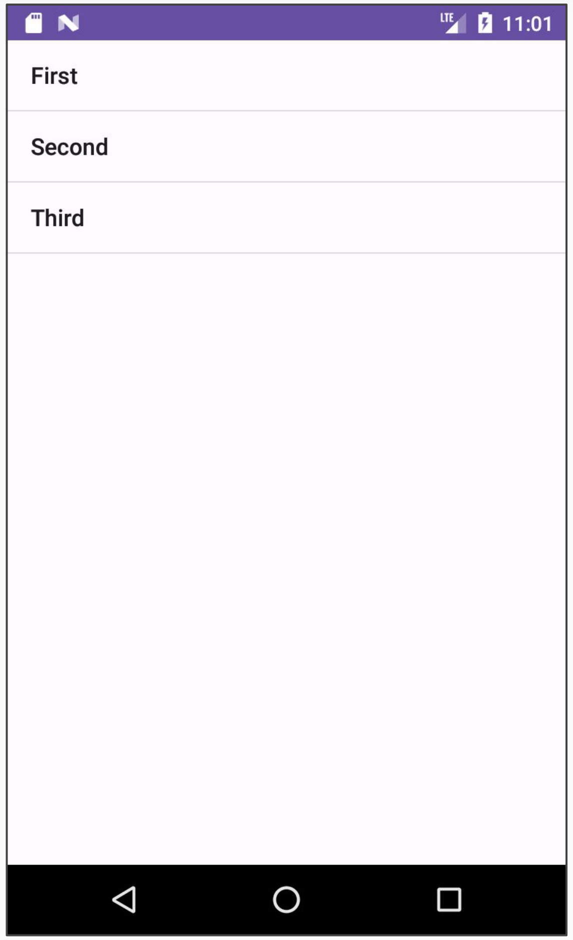
- The context
- A layout to be inflated in the single element of the dynamic layout (i.e. how does a single line of the list look like?)
    - Any android.* is a default layout, in this case hosting a single TextView.
- The data structure that holds the actual data.

```
// Create a list adapter for a string list

val data: Array<String> =
      arrayOf("First", "Second", "Third")
val adapter = ArrayAdapter<String>(
         this,
         android.R.layout.simple_list_item_1,
         data
)
val listView: ListView =
      findViewById(R.id.listView)
listView.adapter = adapter
```

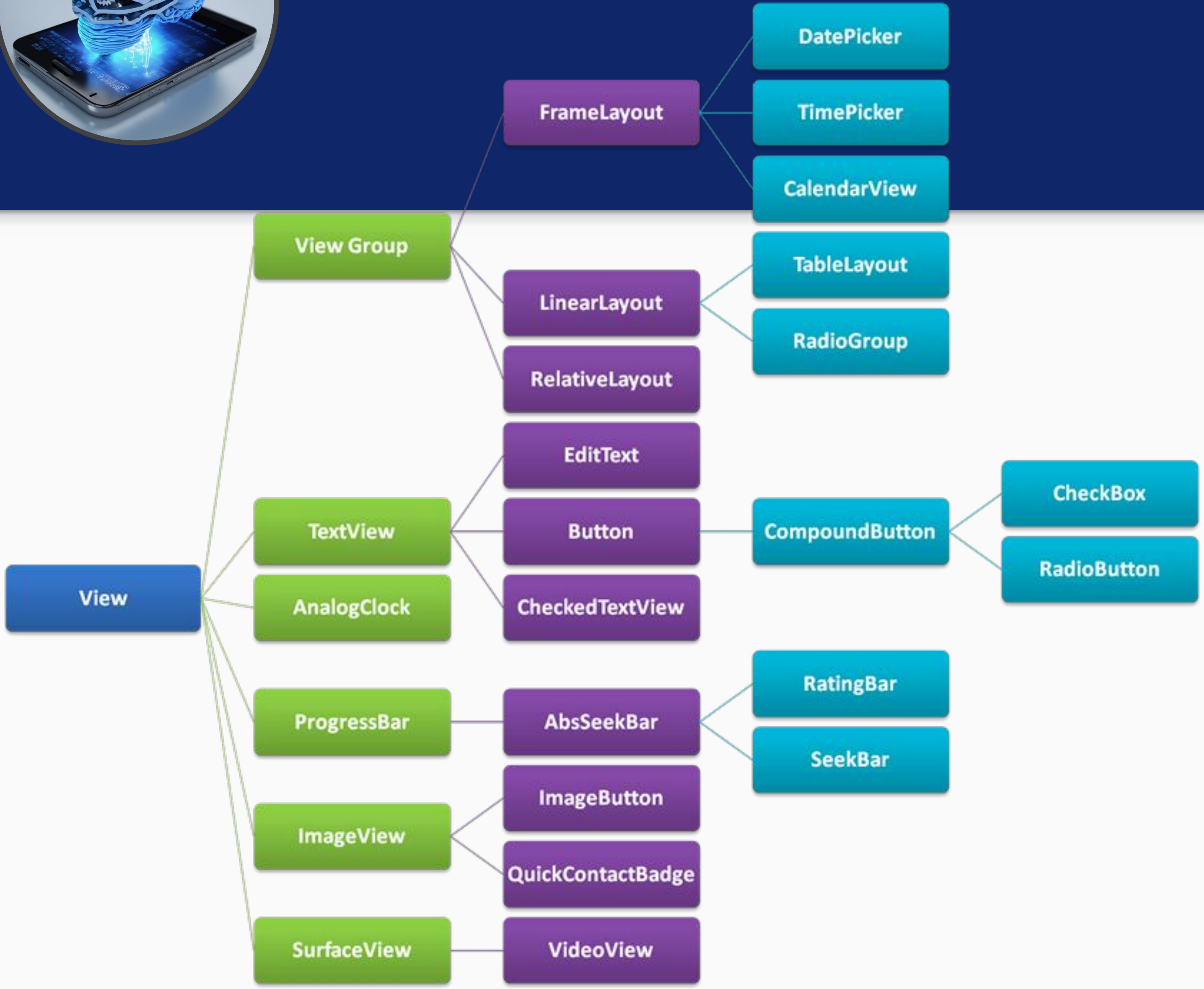
ListView

```
val data: Array<String> = arrayOf("First", "Second", "Third")
val adapter = ArrayAdapter<String>(
        this,
        android.R.layout.simple_list_item_1,
        data
)
val listView: ListView = findViewById(R.id.listView)
listView.adapter = adapter
```

```
<LinearLayout … >
<ListView
      android:id="@+id/listView"
      android:layout_width="match_parent"
      android:layout_height="match_parent" />
</LinearLayout>
```



Widgets



Views are organized in a *hierarchy* of classes.

Widgets are Views with their own behavior implemented.

TextView

XML tags: <TextView>

- Not directly editable by users
- Usually used to display static information
- Methods:
  - public void setText(CharSequence text)
  - public CharSequence getText()
  - public void setSingleLine(boolean singleLine)
  - public void setHorizontallyScrolling(boolean enable)
  - public void setLines(int lines)
  - public void setEllipsize(TextUtils.TruncateAt where)
  - public void setHint(CharSequence hints)

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

- TextUtils.TruncateAt.**END**
- TextUtils.TruncateAt.**MARQUEE**
- TextUtils.TruncateAt.**MIDDLE**
- TextUtils.TruncateAt.**START**

TextView

Simple strings could be linkified automatically.

How? Pick a normal string, and use Linkify.addLinks() to define the kind of links to be created.

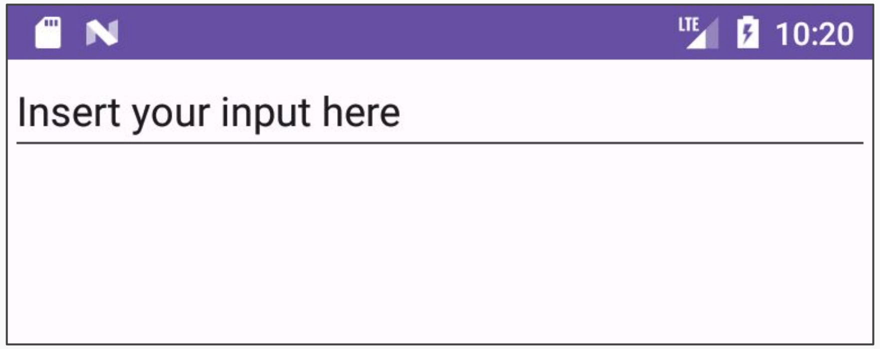
```
Linkify.addLinks(textView,
      Linkify.WEB_URLS or Linkify.EMAIL_ADDRESSES or Linkify.PHONE_NUMBERS )
```

EditText

XML tags: <EditText>

- Similar to a TextView, but editable by the users
- An appropriate keyboard will be displayed

```
<EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:ems="10"
    android:inputType="text"
    android:text="Insert your input here" />
```

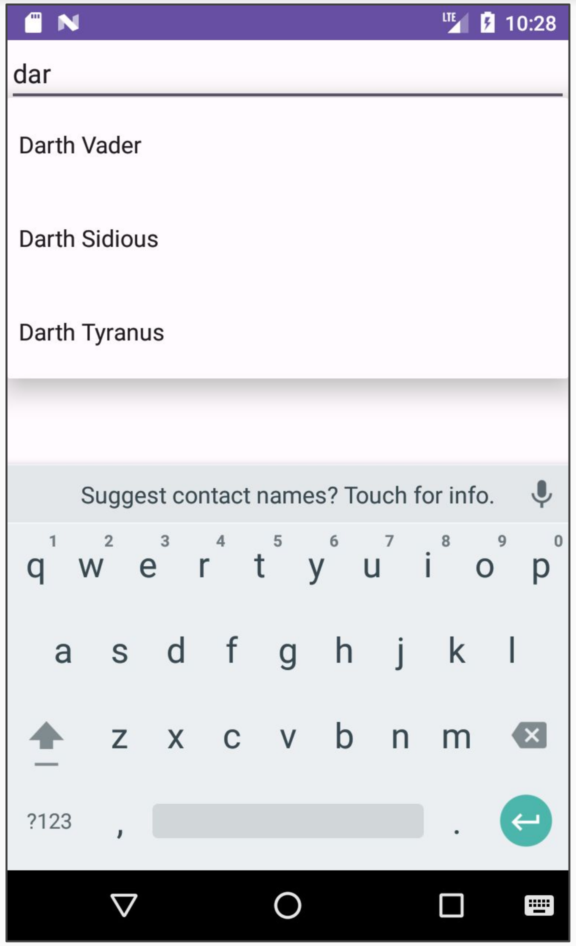


AutoCompleteTextView

XML tags: <AutoCompleteTextView>

- As soon as the user starts typing, hints are displayed
- A list of hints is given through an Adapter

```
val autocompleteTextView: AutoCompleteTextView =
      findViewById(R.id.autoCompleteText)
autocompleteTextView.setAdapter( ArrayAdapter<String> (
      this,
      android.R.layout.simple_dropdown_item_1line,
      arrayOf("Darth Vader", "Darth Sidious", "Darth Tyranus")
))
```



10:28

dar

Darth Vader

Darth Sidious

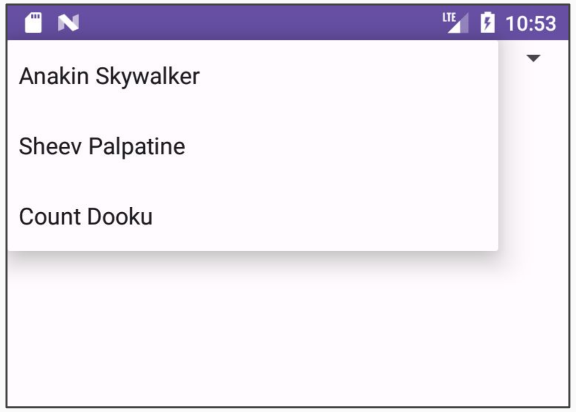
Darth Tyranus

Spinner

XML tags: <Spinner>

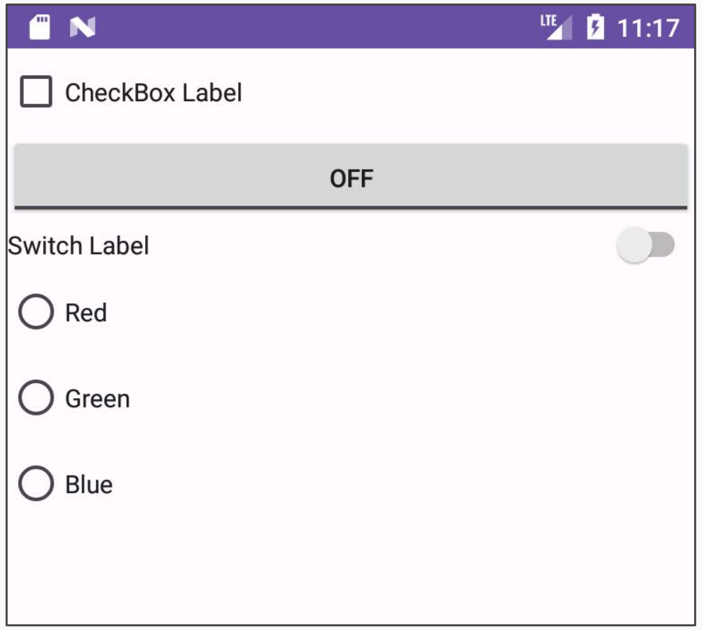
- Provides a quick way to select values from a set
- The value set can be defined in XML (**entries** tag) or through the SpinnerAdapter
- Listener: **OnItemSelectedListener**

```
val spinner: Spinner = findViewById(R.id.spinner)
val spinnerAdapter = ArrayAdapter<String>(
    this, android.R.layout.simple_spinner_item,
    arrayOf("Anakin Skywalker", "Sheev Palpatine", "Count Dooku"))
spinnerAdapter.setDropDownViewResource(
    android.R.layout.simple_spinner_dropdown_item)
spinner.adapter = spinnerAdapter
```



Compound Button

- A subclass of Button
  - represents a Button with a state (ON / OFF)
- It can have different shapes, all technically similar, their difference is in the look-and-feel
  - CheckBox
  - ToggleButton
  - Switch
  - RadioButton
- **isChecked**(): Returns true if the button is checked
- **setChecked**(): Sets the button as checked
- Responds to **OnCheckedChangeListener**

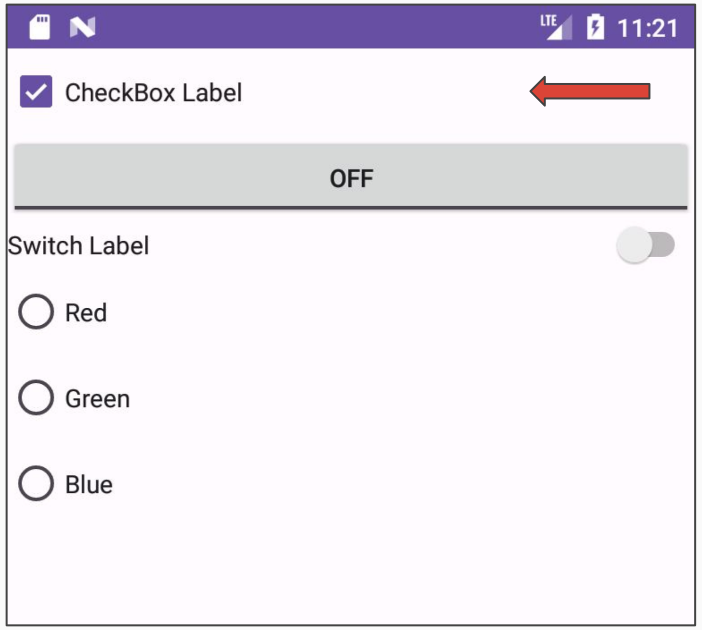


Compound Button

CheckBox XML tags: <CheckBox>

- Used to display binary (or multiple) options for users within a procedure

```
val checkBox: CheckBox = findViewById(R.id.checkBox)
checkBox.setOnCheckedChangeListener{
      _, isChecked -> /* Do your stuff */
}
```

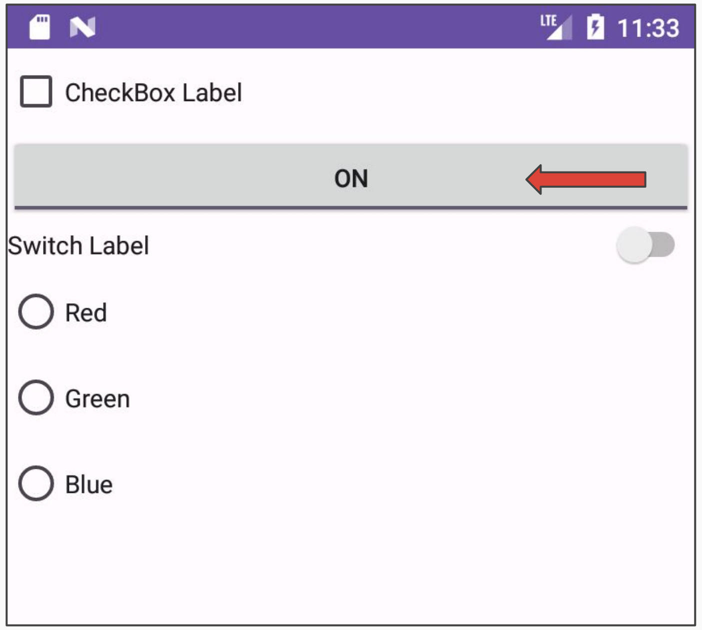


Compound Button

ToggleButton XML tags: <ToggleButton>

- Used to trigger or de-trigger something in real-time

```
val toggleButton: ToggleButton =
    findViewById(R.id.toggleButton)
toggleButton.setOnCheckedChangeListener{
    _, isChecked -> /* Do your stuff */
}
```

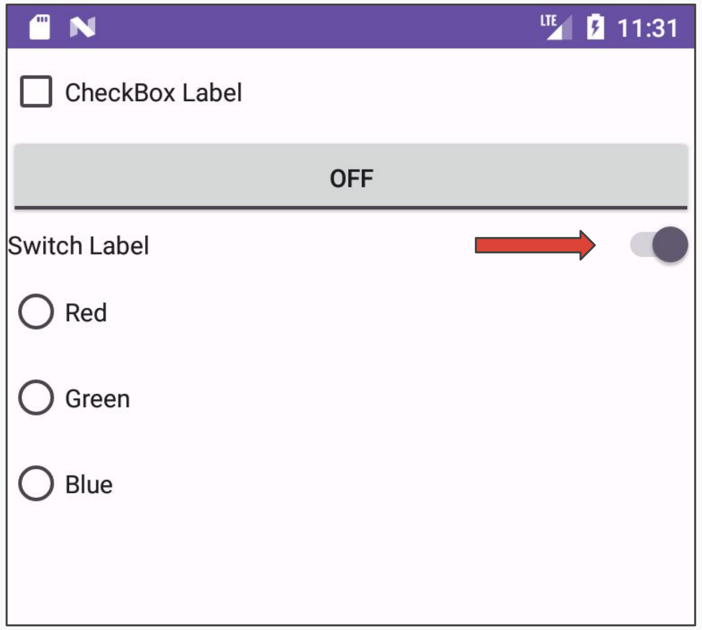


Compound Button

Switch XML tags: <Switch>

- Used to display binary options for users in a settings screen (another option is **<Chip>**)
- Use **SwitchCompat** or **SwitchMaterial** for a modern look

```
val switch: Switch = findViewById(R.id.toggleSwitch)
switch.setOnCheckedChangeListener{
      _, isChecked -> /* Do your stuff */
}
```

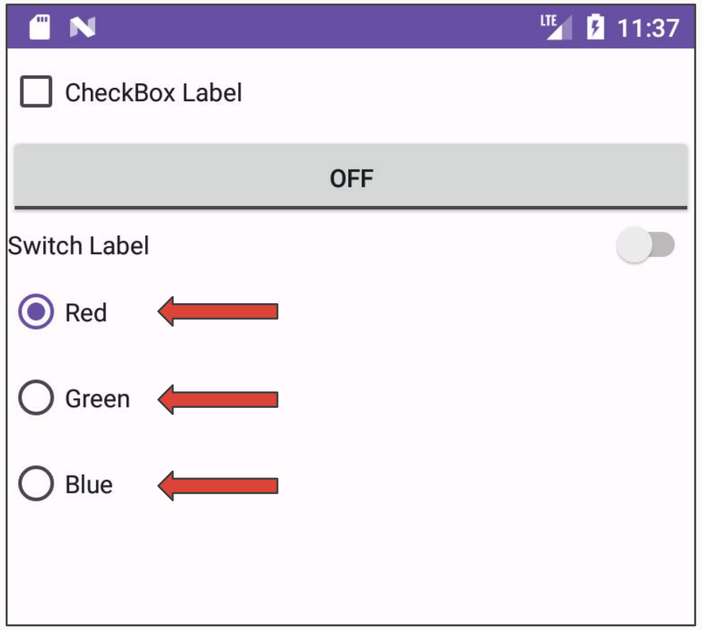


Compound Button

RadioButton XML tags: <RadioButton>

- Enclosed within a **<RadioGroup>** presents a mutually exclusive multiple selection.

```
val radioGroup: RadioGroup = findViewById(R.id.radioGroup)
radioGroup.setOnCheckedChangeListener { _, checkedId ->
    when(checkedId) {
        R.id.radioRed -> /* Do your stuff */
        R.id.radioGreen -> /* Do your stuff */
        R.id.radioBlue -> /* Do your stuff */
        else -> /* Do your stuff */
    }
}
```



RecyclerView

“RecyclerView makes it easy to efficiently display large sets of data. You supply the data and define how each item looks, and the RecyclerView library dynamically creates the elements when they're needed.”

It recycles elements, meaning that when they go off screen, the View is not destroyed, it is instead reused for elements that come on screen.

You can specify the shape of every set member through a dedicated layout.

RecyclerView

Think about it as a highly customizable **ListView**, where you can add, remove and update elements at runtime without redrawing it completely every time something changes:

- with ListView you would call “**notifyDataSetChanged()**”
- here you can do **notifyItemInserted**(), **notifyItemRemoved**(), **notifyItemChanged**() and more…
    - Obviously more efficient when it comes to tens of elements.

RecyclerView

Step 1: define the layout of a single element

- A good idea is to use a **CardView**, which is a styled container that displays data, using elevation and shadow, sticking to a consistent look across the platform.

Here is an example for a list of TODOs

```
<androidx.cardview.widget.CardView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal">
    <TextView
      android:id="@+id/todoTitle"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:textSize="20sp" />
    <CheckBox
      android:id="@+id/todoCheck"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_weight="0"
      android:text="Done?" />
  </LinearLayout>
</androidx.cardview.widget.CardView>
```

RecyclerView

Step 2: define a companion class for the element to hold the data

Our TODO has:

- a title that goes into the TextView
- a boolean value for the CheckBox

```
data class Todo(
   var todoTitle: String,
   var done: Boolean = false
)
```

RecyclerView

Step 3: define a **ViewHolder** which is the runtime container that will get externally inflated with the element layout and then holds the references to its children Views, so they can be customized at runtime.

```
class TodoViewHolder(itemView: View): ViewHolder(itemView) {
   val tvTodoTitle: TextView = itemView.findViewById(R.id.todoTitle)
   val cbDone: CheckBox = itemView.findViewById(R.id.todoCheck)
}
```

RecyclerView

Step 4: define the **Adapter** which takes in input the data (a list of Todo) and generates a ViewHolder for each entry (override RecyclerView.Adapter).

```
class TodoAdapter (private val todos: MutableList<Todo>): Adapter<TodoViewHolder>() {

   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): TodoViewHolder { ... }

   override fun onBindViewHolder(holder: TodoViewHolder, position: Int) { ... }

   override fun getItemCount(): Int { ... }
}
```

RecyclerView

Step 4a: **onCreateViewHolder** invoked when a new element needs to be drawn (we don’t know which one yet). It is expected to return the right ViewHolder, inflated with the right layout. The **parent** is the empty container reserved for this element.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): TodoViewHolder {
    return TodoViewHolder(
        LayoutInflater.from(parent.context).inflate(R.layout.todo_card, parent, false)
    )
}
```

RecyclerView

Step 4b: **onBindViewHolder** invoked when the new element is given a position within the RecyclerView. Here we need to populate the fields of the element.

```
override fun onBindViewHolder(holder: TodoViewHolder, position: Int) {
    holder.apply {
        tvTodoTitle.text = todos[position].todoTitle
        cbDone.apply{
            isChecked = todos[position].done
            setOnCheckedChangeListener { _, b -> todos[position].done = b }
        }
    }
}
```

RecyclerView

Step 4c: **getItemCount** needs to output the number of items in our data struct

```
override fun getItemCount(): Int {
    return todos.size
}
```

We can optionally define some helper functions for adding/removing items

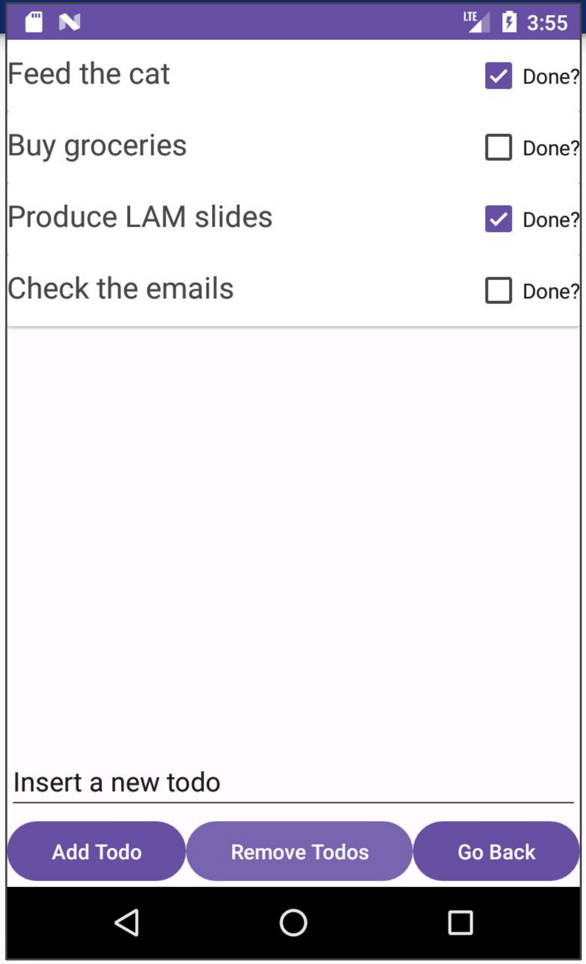
```
fun addTodo(newTodo: Todo) {
    todos.add(newTodo)
    notifyItemInserted(todos.size - 1)
}
```

```
fun deleteDone() {
    todos.removeAll { todo -> todo.done }
    notifyDataSetChanged()
}
```

RecyclerView

Step 5: assign a **LayoutManager** when creating the RecyclerView. It will arrange the items in a defined fashion.

```
val recyclerTodo: RecyclerView =
      findViewById(R.id.recyclerTodo)
recyclerTodo.adapter =
      TodoAdapter(mutableListOf())
recyclerTodo.layoutManager =
       LinearLayoutManager(this)
```



Questions?

federico.montori2@unibo.it