Android: User Interface / Layouts

http://developer.android.com/guide/topics/ui/declaring-layout.html

Ferruccio Damiani

Università di Torino
www.di.unito.it/~damiani

Mobile Device Programming
(Laurea Magistrale in Informatica, a.a. 2017-2018)

Outline

Outline

Widgets/Views

- Layouts are in charge of the placement of the elements on the screen.[1]
- Views are the elements to be placed.
- Widgets are pre-defined, commonly-used View objects.
  - Each Widget has a set of properties defining the UI (e.g., size, colors, layout).
  - Each Widget has a focus and a visibility that the user can modify.

[1] Layouts are subclasses of `ViewGroup`.

Input events [http://developer.android.com/guide/topics/ui/ui-events.html]

The user interacts with the Views and generates events.

- Events for clicks, long clicks, gestures, focus, external events

Input events [http://developer.android.com/guide/topics/ui/ui-events.html]

The user interacts with the Views and generates events.

- Events for clicks, long clicks, gestures, focus, external events

Android manages the creation and distribution of these events, but not the reactions to them: You must implement them by hand: two possible ways:

1. Event Handlers
   - Views have callback methods to handle specific events
     - ⋆ E.g., when a Button is touched, method onTouchEvent() is called
   - Special-purpose reactions are obtained by extending the particular View class and by overriding the method
     - ⋆ Suitable for custom elements
2. Event Listeners
   - Are Interfaces that contain a single callback method
   - This method is called by the Android framework when the event is fired on the particular View

Some Listeners

- OnClickListener: onClick()
- OnLongClickListener: onLongClick()
- OnFocusChangeListener: onFocusChange()
- OnKeyListener: onKey()
- OnCheckedChangeListener: onCheckedChanged()
- OnTouchListener: onTouch()
- OnCreateContextMenuListener: onCreateContextMenu()

Listeners / Option 1

- Implement the callback method
- Define the listener object as an anonymous class
- Pass an instance of the ActionListener implementation to the View through method setOnXXXListener

```
button.setOnClickListener(
    object: View.OnClickListener {
        override fun onClick(view: View?) {
            // Implementation
        }
    }
)
```

→

```
button.setOnClickListener() { view ->
    run {
        // Implementation
    }
}
```

→

```
    button.setOnClickListener() { view ->
        myOnClick(view)
    }
...
fun myOnClick(view: View) {
    // Implementation
}
```

Listeners / Option 2

- Implement the callback method
- Implement the interface in the Activity
- Pass an instance of the listener to the View through method setOnXXXListener

```
1 class MyActivity : AppCompatActivity(), View.OnClickListener {
2     ...
3         button.setOnClickListener(this)
4     ...
5
6     override fun onClick(v: View?) {
7             // Implementation
8     }
```

Listeners / Option 3

- Use onClick in the XML definition of the View (if possible)
- Implement the method in the activity

```xml
<Button
        android:id="@+id/button"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/button"
        android:onClick="myOnClick" />
```

```
fun myOnClick(view: View) {
    // Implementation
}
```

How to fire events in the code

- Through perform"something" methods
  - The corresponding listener (if set) is activated
  - performClick() activates the view's OnClickListener, if defined
- Used to produce events
  - As consequences of other actions
  - To test listeners

Some suggestions

- Avoid displaying too many things
  - Well-known anti-patterns
- Display useful content on your start screen
- Use action bars to provide consistent navigation
- Keep your hierarchies shallow by using horizontal navigation and shortcuts

Outline

Styles and themes

A style is a collection of properties that specify the look and format of a View or window

- Styles share a similar philosophy to cascading stylesheets

A theme is a style applied to an entire Activity or application

- Every View in the Activity or application will apply each style property it supports

- Android provides a large collection of styles and themes
- A reference of all available styles is in the R.style class
- To use the styles listed here, replace all underscores in the style name with a period
  - Theme_NoTitleBar $\rightarrow$ "@android:style/Theme.NoTitleBar"

Example [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/kotlinstyle.git]

ProjectName/res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              ..... >
    ....
    <Button
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:textColor="#00FF00"
              android:typeface="monospace"
              android:text="@string/button_send"
              android:onClick="sendMessage" />
</LinearLayout>
```

The above style can be referenced from an XML layout as @style/CodeFont

```
<Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textColor="#00FF00"
      android:typeface="monospace"
      android:text="@string/button_send"
      android:onClick="sendMessage" />
```

→

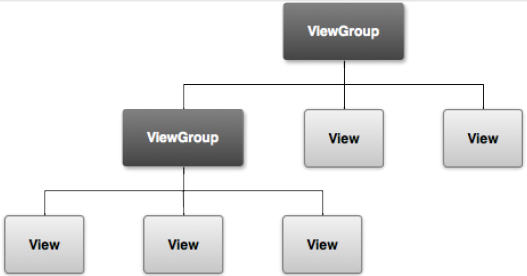
```
<Button
      style="@style/CodeFont"
      android:text="@string/button_send"
      android:onClick="sendMessage" />
```

Outline

The graphical user interface for an Android app is built using a hierarchy of `View` and `ViewGroup` objects.

- `View` objects are usually UI widgets such as **buttons** or **text fields**.
- `ViewGroup` objects are invisible view containers that define how the child views are laid out, such as in a grid or a vertical list.

Android provides an XML vocabulary that corresponds to the subclasses of `View` and `ViewGroup` so you can define your UI in XML using a hierarchy of UI elements.
Layouts are subclasses of the `ViewGroup`.



A layout defines the visual structure for a user interface, such as the UI for an activity or app widget. You can declare a layout in two ways (in the following slides we focus on the first one):

- **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. As done, for instance, in the `java/it.unito.di.educ.pdm18kotlin1/MainActivity.kt` class.
- **Instantiate layout elements at runtime.** Your application can create View and ViewGroup objects (and manipulate their properties) programmatically. As done, for instance, in the `java/it.unito.di.educ.pdm18kotlin1/DisplayMessageActivity.kt` class.

These methods can be used together:

- Declare your application's default layouts in XML.
- Add code that modifies the state of the screen objects.

Example

You could declare your application's default layouts in XML, including the screen elements that will appear in them and their properties. You could then add code in your application that would modify the state of the screen objects, including those declared in XML, at run time.

In general, 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.

- 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, not all vocabulary is identical. In some cases, there are slight naming differences.

Example

The `EditText` element has a text attribute that corresponds to `EditText.setText()`.

Write the 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.

Example

The XML layout below uses a vertical `LinearLayout` to hold a `TextView` and a `Button`:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3               android:layout_width="match_parent"     android:layout_height="match_parent"
4               android:orientation="vertical" >
5     <TextView android:id="@+id/text"     android:layout_width="wrap_content"     android:layout_height="wrap_content"
6               android:text="Hello, I am a TextView" />
7     <Button android:id="@+id/button"     android:layout_width="wrap_content"     android:layout_height="wrap_content"
8             android:text="Hello, I am a Button" />
9 </LinearLayout>
```

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.

Load the XML Resource

When you compile your application, each XML layout file is compiled into a `View` resource.

- You should load the layout resource from your application 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`.

Example

If your XML layout is saved as `main_layout.xml`, you would load it for your Activity like so:

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

The `onCreate()` callback method in your Activity is called by the Android framework when your Activity is launched.

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).
- These attributes are also inherited by any View objects that may extend this class.
- Some attributes are common to all View objects, because they are inherited from the root View class (like the `id` attribute).
- 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.

Attributes / ID

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:

```
1 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).

In order to create views and reference them from the application, 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 = findViewById<Button>(R.id.my_button)
```

→

```
import kotlinx.android.synthetic.main.activity_main.*
...
my_button.YYYY
```

Defining IDs for view objects is important when creating a ConstraintLayout. In a ConstraintLayout, sibling views can define their constraints relative to another sibling view, which is referenced by the unique ID.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

There are a number of other ID resources that are offered by the Android framework.

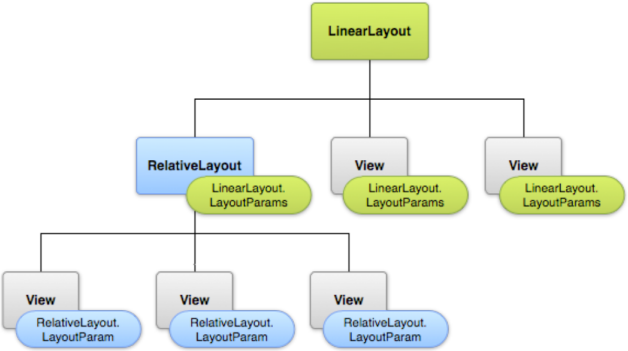
When referencing an Android resource ID, you do not need the plus-symbol, but must add the android package namespace, like so:

```
1 android:id="@android:id/empty"
```

With the android package namespace in place, we're now referencing an ID from the android.R resources class, rather than the local resources class.

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 see below (in the visualization of a view hierarchy with layout parameters associated with each view), the parent view group defines layout parameters for each child view (including the child view group).



All view groups include a width and height (layout_width and layout_height), and each view is required to define them.

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 (named fill_parent before API Level 8) tells your view to become as big as its parent view group will allow.

Best Practice

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 application will display properly across a variety of device screen sizes.

Many LayoutParams also include optional margins and borders.

Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods getLeft() and getTop().

- getLeft() returns the left, or X, coordinate of the rectangle representing the view.
- getTop() returns the top, or Y, coordinate of the rectangle representing the view.

These methods both return the location of the view relative to its parent.

Example

When getLeft() returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

Several convenience methods are offered to avoid unnecessary computations. For instance:

- the methods getRight() and getBottom() return the coordinates of the right and bottom edges of the rectangle representing the view.

Calling getRight() is similar to the following computation: getLeft() + getWidth().

Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possess two pairs of width and height values.

- The *measured width* and *measured height* dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling getMeasuredWidth() and getMeasuredHeight().
- The *width* and *height* (a.k.a. *drawing width* and *drawing height*) define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling getWidth() and getHeight().

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels.

Example

A left padding of 2 will push the view's content by 2 pixels to the right of the left edge.

Padding can be set using the setPadding(int, int, int, int) method and queried by calling getPaddingLeft(), getPaddingTop(), getPaddingRight() and getPaddingBottom().

Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support.

Common Layouts

Each subclass of the ViewGroup class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

Best Practice

Although you can nest one or more layouts within another layout to acheive your UI design, you should strive to keep your layout hierarchy as shallow as possible. Your layout draws faster if it has fewer nested layouts (a wide view hierarchy is better than a deep view hierarchy).

Constraint Layout

Enables you to specify constraints between objects.

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.

Web View

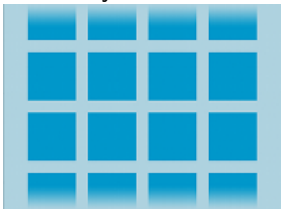
Displays web pages.

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.

Recycler View



Displays a scrolling grid of columns and rows.

Card View

Displays a scrolling single column list of cards.

Outline

Other layout

there are also other layouts but for performance and tooling support reasons they are not recommended:

- Relative Layout
- Frame Layout
- Table Layout
  - For better performance and tooling support, you should instead build your layout with ConstraintLayout.
- Grid View
- List View
  - For better performance in your list, you should instead build your list with RecyclerView.

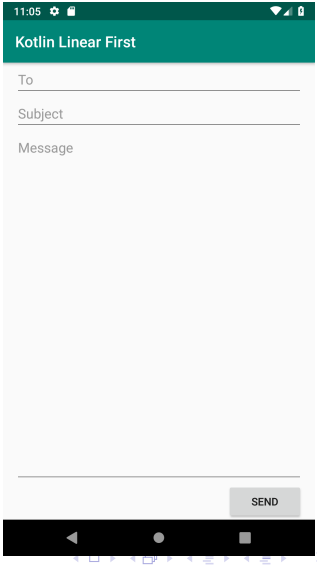
Outline

LinearLayout: overview

- Views presented on a single row/column
  - Defined by layout_orientation
  - Or setOrientation(int orientation)
    - Where orientation is either HORIZONTAL or VERTICAL
- Two further attributes
  - Gravity specifies how to position the child view w.r.t. the parent
  - Weight indicates how much of the extra space in the LinearLayout will be allocated to the view

Example 1 [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinLinearFirst.git]

```
<LinearLayout xmlns:android= ...
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:paddingLeft="@dimen/activity_margin"
       android:paddingRight="@dimen/activity_margin"
       android:orientation="vertical"
       tools:context=".MainActivity" >
       <EditText
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:hint="@string/to" />
      <EditText
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:hint="@string/subject" />
      <EditText
              android:layout_width="fill_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>
```

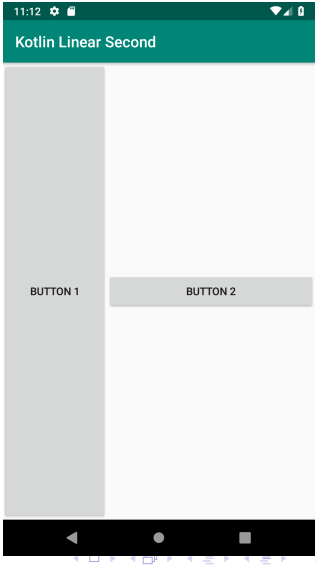


Example 2 [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinLinearSecond.git]

```
<LinearLayout xmlns:android= ...
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:orientation="horizontal"
       tools:context=".MainActivity" >
       <Button
              android:id="@+id/button1"
              android:layout_width="0dp"
              android:layout_height="wrap_content"
              android:layout_weight="1"
              android:text="@string/button1" />
       <Button
              android:id="@+id/button2"
              android:layout_width="0dp"
              android:layout_height="wrap_content"
              android:layout_gravity="center_vertical"
              android:layout_weight="2"
              android:gravity="center_horizontal|center_vertical"
              android:text="@string/button2" />
</LinearLayout>
```

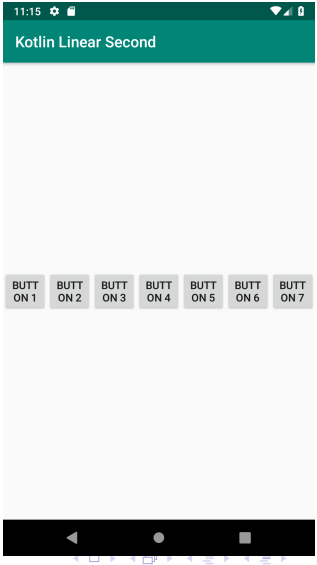
Example 3

```
<LinearLayout xmlns:android= ...
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal"
        tools:context=".MainActivity" >
        <Button
                android:id="@+id/button1"
                android:layout_width="0dp"
                android:layout_height="fill_parent"
                android:layout_weight="1"
                android:text="@string/button1" />
        <Button
                android:id="@+id/button2"
                android:layout_width="0dp"
                android:layout_height="wrap_content"
                android:layout_gravity="center_vertical"
                android:layout_weight="2"
                android:gravity="center_horizontal|center_vertical"
                android:text="@string/button2" />
</LinearLayout>
```



(Counter)Example 4

```
<Button
        android:id="@+id/buttonN"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:layout_weight="1"
        android:text="@string/buttonN" />
```



Outline

Why ConstraintLayout?

[https://developer.android.com/training/constraint-layout/index.html]
[https://developer.android.com/reference/android/support/constraint/ConstraintLayout.html]
[https://developer.android.com/studio/write/layout-editor.html]

Introduced to:

- create large and complex layouts with a flat view hierarchy (no nested view groups)[2]
- overcome the performance problems of other layouts (e.g., nested weighted Linear Layouts)

It:

- allows us to lay out child views using 'constraints' to define position based relationships between different views found in our layout
- is similar to RelativeLayout in that all views are laid out according to relationships between sibling views and the parent layout
- is more flexible than RelativeLayout and easier to use with Android Studio's Layout Editor

[2] Flat hierarchies display quick and are memory efficient

How to define a ConstraintLayout?

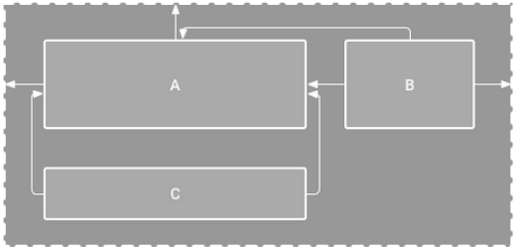
A ConstraintLayout is a ViewGroup which allows you to position and size widgets in a flexible way.

- To define a view's position, 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
  - 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[3]

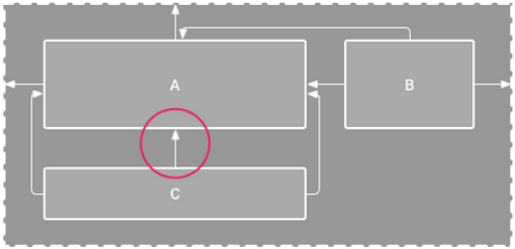
[3] Often more are necessary

Layout Editor

In the left figure, 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

Although a missing constraint won't cause a compilation error, the Layout Editor indicates missing constraints as an error in the toolbar. To view the errors and other warnings, click Show Warnings and Errors (!). To help you avoid missing constraints, the Layout Editor can automatically add constraints for you with the Autoconnect and infer constraints features.

Types of constraints

There are currently various types of constraints that you can use:

- Relative positioning
- Margins
- Centering positioning
- Circular positioning
- Visibility behavior
- Dimension constraints
- Chains
- Virtual Helpers objects

Note that you cannot have a circular dependency in constraints.

Layout Editor's visual tools: Types of layout behavior

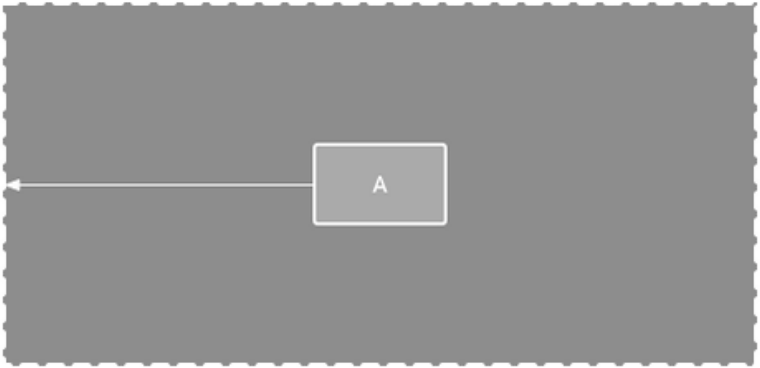
You can use constraints to achieve different types of layout behavior, as described in the following:

1. Parent position
2. Order position
3. Alignment
4. Baseline alignment
5. Constrain to a guideline
6. Constrain to a barrier

1. Parent position

Constrain the side of a view to the corresponding edge of the layout.

In the figure, 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.

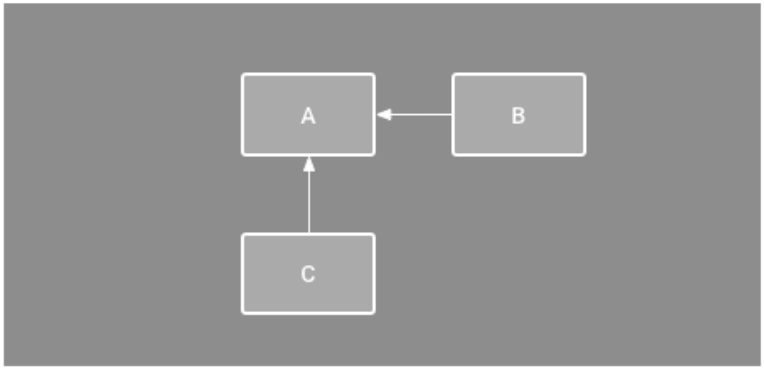


A horizontal constraint to the parent

2. Order position

CDefine the order of appearance for two views, either vertically or horizontally.

In the figure, 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.



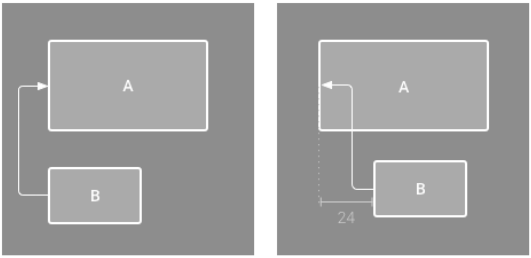
A horizontal and vertical constraint

3. Alignment

Align the edge of a view to the same edge of another view.

In the figure, 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 7 shows B with a 24dp offset alignment. The offset is defined by the constrained view's margin.
You can also select all the views you want to align, and then click Align in the toolbar to select the alignment type.



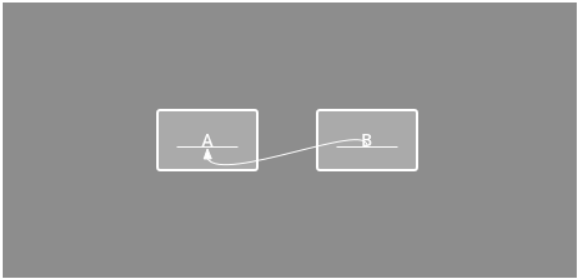
Left: A horizontal alignment constraint. **Right: An offset horizontal alignment constraint**

4. Baseline alignment

Align the text baseline of a view to the text baseline of another view.

In the figure, the first line of B is aligned with the text in A.

To create a baseline constraint, select the text view you want to constrain and then click Edit Baseline, which appears below the view. Then click the text baseline and drag the line to another baseline.

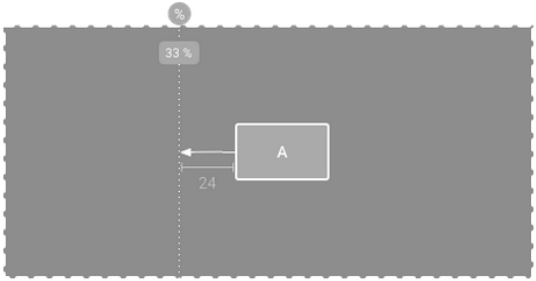


A baseline alignment constraint

5. 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.

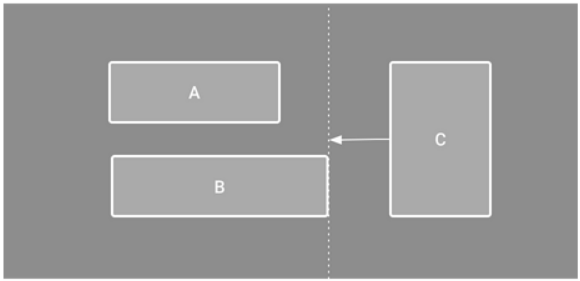


A view constrained to a guideline

6. Constrain to a barrier

Similar to a guideline, a barrier is an invisible line that you can constrain views to. Except a barrier does not define its own position; instead, the barrier position moves based on the position of views contained within it. This is useful when you want to constrain a view to the a set of views rather than to one specific view.

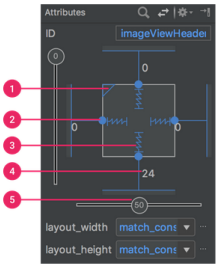
In the figure, view C is constrained to the right side of a barrier. The barrier is set to the "end" (or the right side in a left-to-right layout) of both view A and view B. So the barrier moves depending on whether the right side of view A or view B is is farthest right.



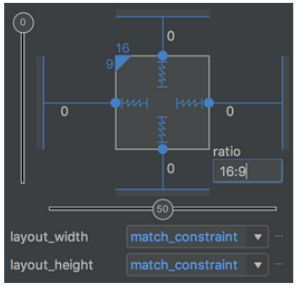
View C is constrained to a barrier, which moves based on the position/size of both view A and view B

Adjust the view size

You can use the corner handles to resize a view, but this hard codes the size so the view will not resize for different content or screen sizes. To select a different sizing mode, click a view and open the Attributes window on the right side of the editor.
Near the top of the Attributes window is the view inspector, which includes controls for several layout attributes, as shown in the figure (this is available only for views in a constraint layout).
You can change the way the height and width are calculated by clicking the symbols indicated with callout 3 in the figure.



The **Attributes** window includes controls for 1 size ratio, 2 delete constraint, 3 height/width mode, 4 margins, and 5 constraint bias.

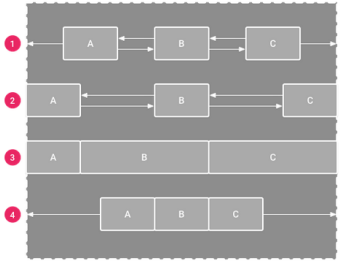


The view is set to a 16:9 aspect with the width based on a ratio of the height.

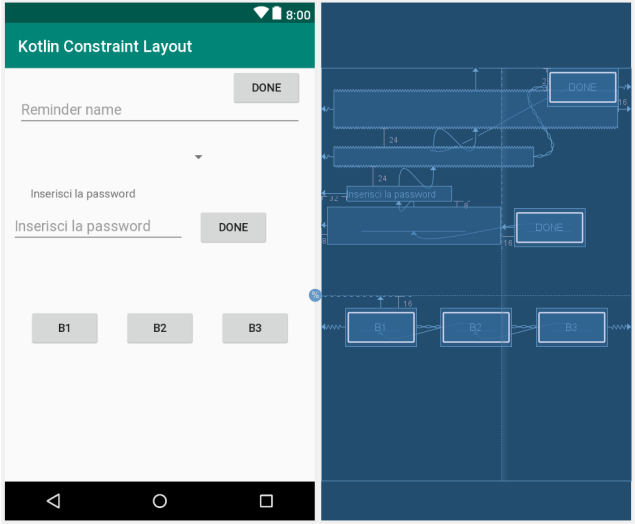
Control linear groups with a chain

A chain is a group of views that are linked to each other with bi-directional position constraints.
A chain allows you to distribute a group of views horizontally or vertically with the following styles (as shown in the figure):

1. **Spread:** The views are evenly distributed (after margins are accounted for). This is the default.
2. **Spread inside:** The first and last view are affixed to the constraints on each end of the chain and the rest are evenly distributed.
3. **Weighted:** When the chain is set to either spread or spread inside, you can fill the remaining space by setting one or more views to 'match constraints" (0dp).
4. **Packed:** The views are packed together (after margins are accounted for). You can then adjust the whole chain's bias (left/right or up/down) by changing the chain's head view bias.



Example [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinConstraintLayout.git]



Outline

WebView [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinWebView.git]

The WebView class is an extension of Android's View class.

- Allows you to display web pages as a part of your activity layout.
  - It does not include any features of a fully developed web browser, such as navigation controls or an address bar.
  - All that WebView does, by default, is show a web page.
- You
  - Must enable JavaScript if needed.
  - Can create interfaces between JavaScript code and Android code.
  - Can control navigation.

```
1 webview.apply {
2     loadUrl("http://magistrale.educ.di.unito.it/")
3     settings.javaScriptEnabled = true
4 }
5 ...
```

Outline

RecycleView

https://developer.android.com/guide/topics/ui/layout/recyclerview

The RecyclerView widget is a more advanced and flexible version of ListView.

The RecyclerView fills itself with views provided by a layout manager that you provide. You can use one of standard layout managers (LinearLayoutManager or GridLayoutManager), or implement your own.

The views in the list are represented by view holder objects. The RecyclerView creates only as many view holders as are needed to display the on-screen portion of the dynamic content, plus a few extra. As the user scrolls through the list, the RecyclerView takes the off-screen views and rebinds them to the data which is scrolling onto the screen.

The view holder objects are managed by an adapter. The adapter creates view holders as needed. The adapter also binds the view holders to their data. It does this by assigning the view holder to a position, and calling the adapter's onBindViewHolder() method. That method uses the view holder's position to determine what the contents should be, based on its list position.

RecycleView - How to create it [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinRecycleView.git]

- Add the support library

  Open the build.gradle file for your app module and add the support library:

```
dependencies {
    ...
    implementation com.android.support:recyclerview-v7:28.0.0
}
```

- Add RecyclerView to your layout

  Now you can add the RecyclerView to your layout file, activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".MainActivity"
        android:id="@+id/lista"
        android:layout_marginLeft="16dp"
        android:layout_marginRight="16dp"
        tools:listitem="@layout/element" />
```

RecycleView - How to create it [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinRecycleView.git]

- Connect the RecycleView to a layout manager and attach an adapter for the data to be displayed: MainActivity.kt:

```
1   private val mColumnCount = 1
2   private val enableStaggeredGrid = false
3   private val staggeredGridOrientation = StaggeredGridLayoutManager.VERTICAL
4   private lateinit var viewAdapter: RecyclerView.Adapter<*>
5   private lateinit var viewManager: RecyclerView.LayoutManager
6
7   override fun onCreate(savedInstanceState: Bundle?) {
8       ...
9       if (mColumnCount <= 1) {
10          viewManager = LinearLayoutManager(this)
11      } else {
12          if (enableStaggeredGrid) {
13              viewManager = StaggeredGridLayoutManager(mColumnCount, staggeredGridOrientation)
14          } else {
15              viewManager = GridLayoutManager(this, mColumnCount)
16      } }
17      viewAdapter = MyElementRecyclerViewAdapter(DummyList.getLista())
18      lista.apply {
19          layoutManager = viewManager
20          adapter = viewAdapter
21      }
22  }
```

RecycleView - How to create it [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinRecycleView.git]

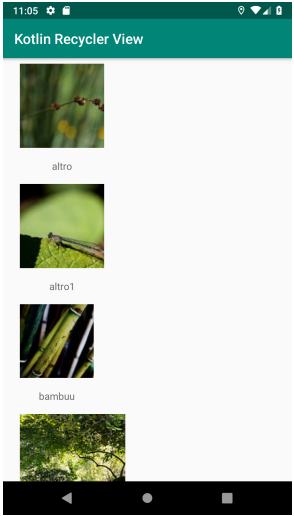
- Define the adapter to use:
  MyElementRecyclerViewAdapter.kt:

```
1  class MyElementRecyclerViewAdapter(private val mValues: List<DummyList.DummyItem>) :
2      RecyclerView.Adapter<MyElementRecyclerViewAdapter.ViewHolder>() {
3
4      // Provide a reference to the views for each data item
5      class ViewHolder(val mView: View) : RecyclerView.ViewHolder(mView)
6
7      // Return the size of your dataset (invoked by the layout manager)
8      override fun getItemCount() = mValues.size
9
10     // Create new views (invoked by the layout manager)
11     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MyElementRecyclerViewAdapter.ViewHolder {
12         val view = LayoutInflater.from(parent.context).inflate(R.layout.element, parent, false) as View
13         return ViewHolder(view)
14     }
15
16     // Replace the contents of a view (invoked by the layout manager)
17     override fun onBindViewHolder(holder: MyElementRecyclerViewAdapter.ViewHolder, position: Int) {
18         holder.mView.apply {
19             nome.text = mValues[position].titolo
20             imageView.setBackgroundResource(mValues[position].id)
21         }
22     }
23 }
```

RecycleView - How to create it [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinRecycleView.git]

Other resources:

- DummyList.kt:
  Contains the list of items to display
- element.xml:
  Contains the layout for displaying a sigle element
- The standard layout manager:
  - LinearLayoutManager: arranges the items in a one-dimensional list.
  - GridLayoutManager: arranges the items in a two-dimensional grid, like the squares on a checkerboard.
  - StaggeredGridLayoutManager: arranges the items in a two-dimensional grid, with each column slightly offset from the one before.
- res\raw:
  The list of images to display



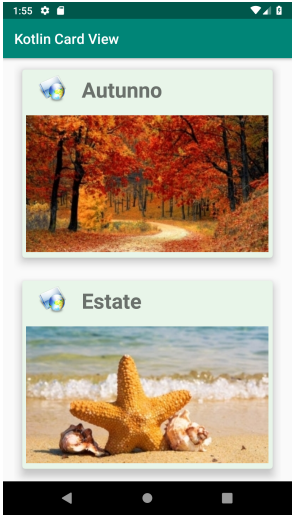
Outline

CardView [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/KotlinCardView.git]

A card is a sheet of material that may contain a photo, text, and a link about a single subject. They may display content containing elements of varying size, such as photos with captions of variable length.

Cards contain content and actions about a single subject. They can be used standalone, or as part of a list. Cards are meant to be interactive, and aren't meant to be be used solely for style purposes.

- app module's build.gradle file: add the new dependence for cards
- element.xml: design of a single card
- Additional information on Cards of Material Design: `https://material.io/design/components/cards.html`



Outline

Re-using Layouts with <include/>

http://developer.android.com/training/improving-layouts/reusing-layouts.html

To efficiently re-use complete layouts, you can use the <include/> and <merge/> tags to embed another layout inside the current layout.

- Create a new XML file and define the layout
  - Example: a title bar that should be included in each activity
- Add the <include/> tag inside the new layout
- The <merge/> tag helps eliminate redundant view groups in your view hierarchy
  - When you include this layout in another layout, the system ignores the <merge/> element and places the elements directly in the layout

Create a Re-usable Layout

If you already know the layout that you want to re-use, create a new XML file and define the layout. For example, here's a layout from the G-Kenya codelab that defines a title bar to be included in each activity (titlebar.xml):

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/titlebar_bg">

    <ImageView android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:src="@drawable/gafricalogo" />
</FrameLayout>
```

The root View should be exactly how you'd like it to appear in each layout to which you add this layout.

Use the <include/> Tag

Inside the layout to which you want to add the re-usable component, add the <include/> tag. For example, here's a layout file from the G-Kenya codelab that includes the title bar from the above slide:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/app_bg"
    android:gravity="center_horizontal">

    <include layout="@layout/titlebar"/>

    <TextView android:layout_width="match_parent"
              android:layout_height="wrap_content"
              android:text="@string/hello"
              android:padding="10dp" />

    ...

</LinearLayout>
```

You can also override all the layout parameters (any android:layout_* attributes) of the included layout's root view by specifying them in the <include/> tag. For example:

```
<include android:id="@+id/news_title"
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         layout="@layout/title"/>
```

However, if you want to override layout attributes using the <include/> tag, you must override both android:layout_height and android:layout_width in order for other layout attributes to take effect.

Use the <merge> Tag

The <merge> tag helps eliminate redundant view groups in your view hierarchy when including one layout within another.

Example

If your main layout is a vertical LinearLayout in which two consecutive views can be re-used in multiple layouts, then the re-usable layout in which you place the two views requires its own root view. However, using another LinearLayout as the root for the re-usable layout would result in a vertical LinearLayout inside a vertical LinearLayout. The nested LinearLayout serves no real purpose other than to slow down your UI performance.

To avoid including such a redundant view group, you can instead use the <merge> element as the root view for the re-usable layout. For example:

```
<merge xmlns:android="http://schemas.android.com/apk/res/android">
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/add"/>
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/delete"/>
</merge>
```

Now, when you include this layout in another layout (using the <include/> tag), the system ignores the <merge> element and places the two buttons directly in the layout, in place of the <include/> tag.

Outline

RelativeLayout: overview

- 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)
  - Relative to the parent area (such as aligned to the bottom, left or center)
- Useful to align views
- Can eliminate nested view groups and keep your layout hierarchy flat

Positioning Views

By default, all child views are drawn at the top-left of the layout, so you must define the position of each view using the various layout properties available from `RelativeLayout.LayoutParams`.

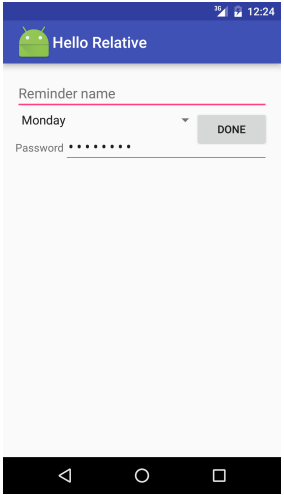
Some of the many layout properties available to views in a `RelativeLayout` include:

- android:layout_alignParentTop
  - If "true", makes the top edge of this view match the top edge of the parent.
- android:layout_centerVertical
  - If "true", centers this child vertically within its parent.
- android:layout_below
  - Positions the top edge of this view below the view specified with a resource ID.
- android:layout_toRightOf
  - Positions the left edge of this view to the right of the view specified with a resource ID.

In your XML layout, dependencies against other views in the layout can be declared in any order.

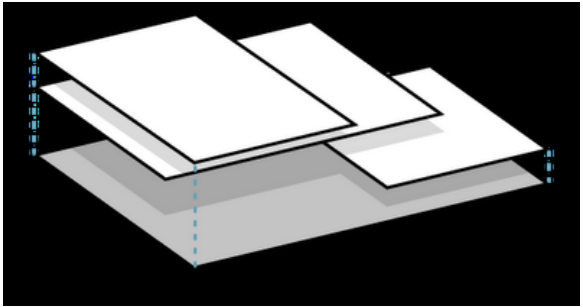
Example [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/HelloRelative.git]

```
<RelativeLayout xmlns:android= ... >

      <EditText
            android:id="@+id/name" ...
            android:hint="@string/reminder" />
       <Spinner
            android:id="@+id/options" ...
            android:layout_below="@id/name"
            android:layout_alignParentLeft="true"
            android:layout_toLeftOf="@+id/ok"
            android:entries="@array/days_of_week" />
      <EditText
            android:id="@+id/password" ...
            android:inputType="textPassword"
            android:layout_below="@id/options"
            android:layout_alignParentRight="true"
            android:layout_toRightOf="@+id/passwordLabel" />
      <TextView
            android:id="@id/passwordLabel" ...
            android:layout_alignBaseline="@id/password" />
      <Button
            android:id="@id/ok" ...
            android:layout_below="@id/name"
            android:layout_alignParentRight="true" />
</RelativeLayout>
```



FrameLayout

- Designed to display a single or multiple UI elements
  - The position of multiple children can be controlled by assigning a gravity to each child
  - Elements that overlap are displayed overlapping
- Child views are drawn in a stack, with the most recently added child on top
- Adds android:visibility to manage the visibility of views



Example [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/HelloFrame.git]

```
<FrameLayout xmlns:android= ...
        android:layout_width="match_parent"
        android:layout_height="match_parent" >

        <ImageView
                android:src="@drawable/main_logo"
                android:contentDescription="@string/key"
                android:scaleType="fitCenter"
                android:layout_height="fill_parent"
                android:layout_width="fill_parent"/>
        <TextView
                android:text="@string/test"
                android:textSize="30sp"
                android:textStyle="bold"
                android:textColor="@color/colorText"
                android:layout_height="fill_parent"
                android:layout_width="fill_parent"
                android:gravity="center"/>
</FrameLayout>
```

TableLayout

- TableLayout is a ViewGroup that displays child View elements in 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 columns (as they can in HTML).
- TableRow objects are the child views of a TableLayout (each TableRow defines a single row in the table).
  - Each row has zero or more cells, each of which is defined by any kind of other View.
  - So, the cells of a row may be composed of a variety of View objects, like ImageView or TextView objects.
  - A cell may also be a ViewGroup object (for example, you can nest another TableLayout as a cell).

Example

1. Start a new project named HelloTableLayout (Empty Activity).
2. Open the res/layout/main.xml file and insert the XML in the next slide.
3. Make your HelloTableLayout Activity load this layout in the onCreate() method:

```
1 public void onCreate(Bundle savedInstanceState) {
2     super.onCreate(savedInstanceState);
3     setContentView(R.layout.main);
4 }
```

4. Run the application. You should see the following:

[git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/HelloTable.git]

res/layout/main.xml

```
<TableLayout android:stretchColumns="1"
        ... >
  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="Open..."
      android:padding="3dp" />
    <TextView
      android:text="Ctrl-O"
      android:gravity="right"
      android:padding="3dp" />
  </TableRow>
  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="Save..."
      android:padding="3dp" />
    <TextView
      android:text="Ctrl-S"
      android:gravity="right"
      android:padding="3dp" />
  </TableRow>
```

```

  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="Save As..."
      android:padding="3dp" />
    <TextView
      android:text="Ctrl-Shift-S"
      android:gravity="right"
      android:padding="3dp" />
  </TableRow>
  <TableRow>
    <View android:layout_span="3"
      android:layout_height="2dp"
      android:background="#FF909090" />
  </TableRow>
  <TableRow>
    <TextView
      android:text="X"
      android:padding="3dp" />
    <TextView
      android:text="Import..."
      android:padding="3dp" />
  </TableRow>
```

```

  <TableRow>
    <TextView
      android:text="X"
      android:padding="3dp" />
    <TextView
      android:text="Export..."
      android:padding="3dp" />
    <TextView
      android:text="Ctrl-E"
      android:gravity="right"
      android:padding="3dp" />
  </TableRow>
  <TableRow>
    <View android:layout_span="3"
      android:layout_height="2dp"
      android:background="#FF909090" />
  </TableRow>
  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="Quit"
      android:padding="3dp" />
  </TableRow>

</TableLayout>
```

Lists and grids

- When the content for a Layout is dynamic or not predetermined, we need a layout that subclasses AdapterView to populate the layout with Views at runtime
- Items are automatically inserted into the Layout by an Adapter
  - It pulls content from a source such as an array or database query
  - Converts each item result into a View that is placed into the Layout
- ListView displays a list of scrollable items
- GridView displays items in a two-dimensional, scrollable grid

Adapter

- Used to visualize data
  - Acts as a bridge between an AdapterView and the underlying data for that view
    - ListView, GridView, Spinner are subclasses of AdapterView query
  - Provides access to the data items
- Makes a ViewGroup to interact with data
  - Also responsible for making a View for each item in the data set
- Some methods
  - isEmpty(), getItem(), getCount(), getView()

A couple of Adapters

1. ArrayAdapter should be used when the data source is an array
   - Acts as a bridge between an AdapterView and the underlying data for that view
   - It creates a view for each array item by calling toString() on each item and placing the contents in a TextView
2. SimpleCursorAdapter should be used when data come from a Cursor
   - We must specify a layout for each row in the Cursor and specify the columns that should be inserted into each view of the layout

onItemClickListener should be used to respond to click events on each item in an AdapterView

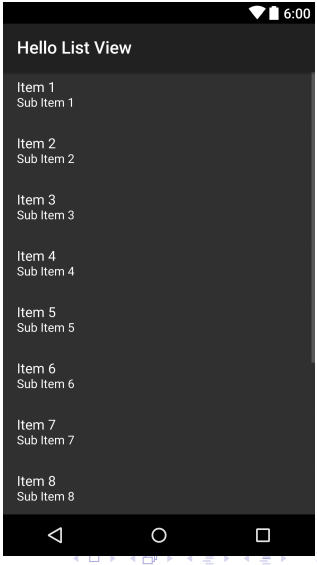
Example (ListView) [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/HelloListView.git]

1. Start a new project named Hello List View (Empty Activity).

2. Open the res/layout/activity_main.xml file and insert the following XML:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <ListView xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent"
5     android:orientation="vertical"
6     android:id="@+id/list" />
```

The "Design" rendering in AndroidStudio should be as shown on the right.



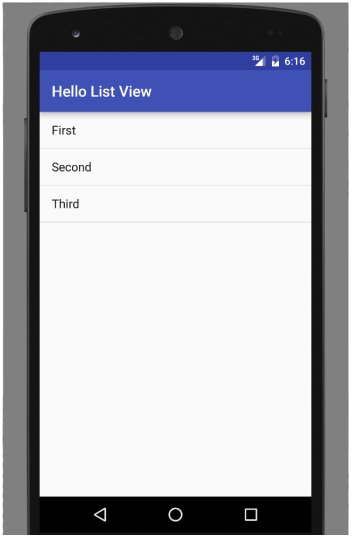
3 Create the res/layout/simple_list_item_1 file and insert the following XML:

```xml
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    style="?android:attr/textAppearanceLarge"
    android:paddingTop="2dip"
    android:paddingBottom="3dip"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

4 Open the MainActivity.java file and modify the onCreate() method as follows:

```java
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String[] data = {"First", "Second", "Third"};
        ListView lv = (ListView) findViewById(R.id.list);
        lv.setAdapter(new ArrayAdapter<String>(this,
                android.R.layout.simple_list_item_1, data));
    }
```

5 Run the application. You should see the screen on the right.



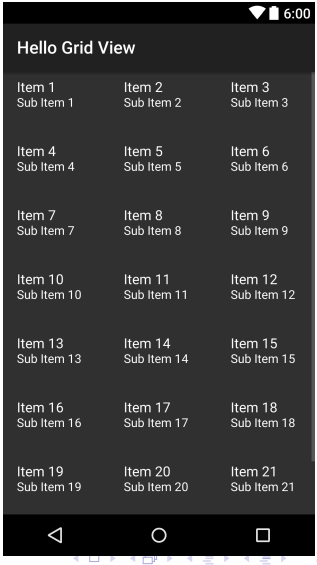
Example (GridView) [git clone https://<login>@gitlab2.educ.di.unito.it/ProgMob/HelloGridView.git]

1. Start a new project named HelloGridView (Empty Activity).

2. Open the res/layout/activity_main.xml file and insert the following XML:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <GridView xmlns:android="http://schemas.android.com/apk/res/android"
3          android:id="@+id/gridview"
4          android:layout_width="fill_parent"
5          android:layout_height="fill_parent"
6          android:columnWidth="90dp"
7          android:numColumns="auto_fit"
8          android:verticalSpacing="10dp"
9          android:horizontalSpacing="10dp"
10         android:stretchMode="columnWidth"
11         android:gravity="center"
12 />
```

The "Design" rendering in AndroidStudio should be as shown on the right.



3 Open the MainActivity.java file and modify the onCreate() method as follows:

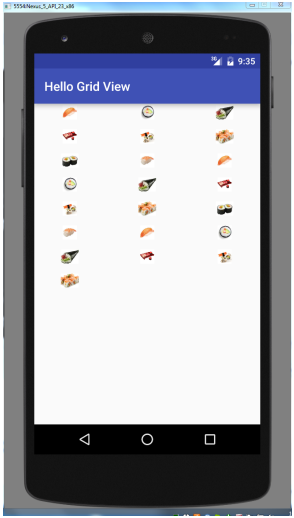
```
1    public void onCreate(Bundle savedInstanceState) {
2        super.onCreate(savedInstanceState);
3        setContentView(R.layout.activity_main);
4        GridView gridview = (GridView) findViewById(R.id.gridview);
5        gridview.setAdapter(new ImageAdapter(this));
6        gridview.setOnItemClickListener(new AdapterView.OnItemClickListener() {
7            public void onItemClick(AdapterView<?> parent,
8                                    View v, int position, long id) {
9                Toast.makeText(MainActivity.this, "" + position,
10                       Toast.LENGTH_SHORT).show();
11           }
12       });
13   }
```

- After the main.xml layout is set for the content view, the GridView is captured from the layout with findViewById(int).
- The setAdapter() method then sets a custom adapter (ImageAdapter) as the source for all items to be displayed in the grid. The ImageAdapter is created in the next step.
- To do something when an item in the grid is clicked, the setOnItemClickListener() method is passed a new AdapterView.OnItemClickListener. This anonymous instance defines the onItemClick() callback method to show a Toast that displays the index position (zero-based) of the selected item (in a real world scenario, the position could be used to get the full sized image for some other task).

4 Add to the MainActivity.java file the ImageAdapter class given below.

```
class ImageAdapter extends BaseAdapter {
    private Context mContext;

    // references to our images
    private Integer[] mThumbIds = {
            R.drawable.sample_2, R.drawable.sample_3, R.drawable.sample_4, R.drawable.sample_5,
            R.drawable.sample_6, R.drawable.sample_7, R.drawable.sample_0, R.drawable.sample_1,
            R.drawable.sample_2, R.drawable.sample_3, R.drawable.sample_4, R.drawable.sample_5,
            R.drawable.sample_6, R.drawable.sample_7, R.drawable.sample_0, R.drawable.sample_1,
            R.drawable.sample_2, R.drawable.sample_3, R.drawable.sample_4, R.drawable.sample_5,
            R.drawable.sample_6, R.drawable.sample_7
    };

    public ImageAdapter(Context c) {
        mContext = c;
    }

    public int getCount() {
        return mThumbIds.length;
    }

    public Object getItem(int position) {
        return null;
    }

    public long getItemId(int position) {
        return 0;
    }
```

```
    // create a new ImageView for each item referenced by the Adapter
    public View getView(int position, View convertView, ViewGroup parent) {
        ImageView imageView;
        if (convertView == null) {
            // if it's not recycled, initialize some attributes
            imageView = new ImageView(mContext);
            imageView.setLayoutParams(new GridView.LayoutParams(85, 85));
            imageView.setScaleType(ImageView.ScaleType.CENTER_CROP);
            imageView.setPadding(8, 8, 8, 8);
        } else {
            imageView = (ImageView) convertView;
        }

        imageView.setImageResource(mThumbIds[position]);
        return imageView;
    }

} // end of class ImageAdapter
```



5 Add to res/drawable the sample_0.jpg,...,sample_7.jpg files

6 Run the application. You should see the screen on the right. Then, click on the fourth image and you will see...