USER INTERFACE

ROMI FADILLAH RAHMAT

VIEW AND VIEW GROUPS

- Every item in a user interface is a subclass of the Android *View* class (to be precise *android.view.View*).
- The Android SDK provides a set of pre-built views that can be used to construct a user interface.
- Typical examples include standard items such as the Button, CheckBox, ProgressBar and TextView classes. Such views are also referred to as *widgets* or *components*.
- For requirements that are not met by the widgets supplied with the SDK, new views may be created either by subclassing and extending an existing class, or creating an entirely new component by building directly on top of the View class.

VIEW GROUPS

- A view can also be comprised of multiple other views (otherwise known as a *composite view*). Such views are subclassed from the Android *ViewGroup* class (*android.view.ViewGroup*) which is itself a subclass of *View*.
- An example of such a view is the RadioGroup, which is intended to contain multiple RadioButton objects such that only one can be in the "on" position at any one time. In terms of structure, composite views consist of a single parent view (derived from the ViewGroup class and otherwise known as a *container view* or *root element)* that is capable of containing other views (known as *child views*).
- Another category of ViewGroup based container view is that of the layout manager

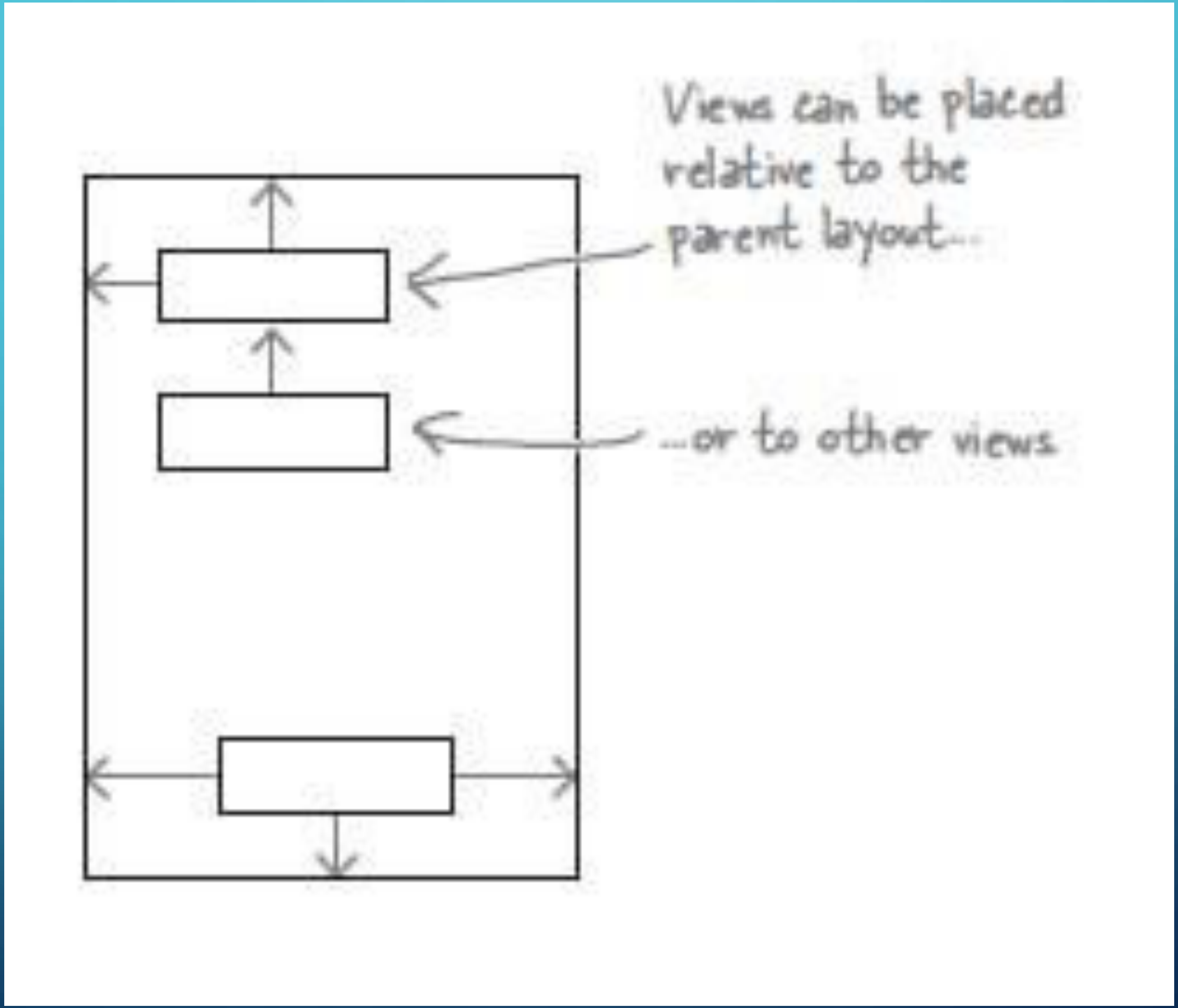
LAYOUT

- Layout defines what a screen looks like -> defined using XML.
- Type of layout :
  - RelativeLayout
  - LinearLayout
  - GridLayout
  - TableLayout
  - FrameLayout
  - AbsoluteLayout
  - ConstraintLayout – Introduced in Android 7 (talk about it later)

RELATIVELAYOUT

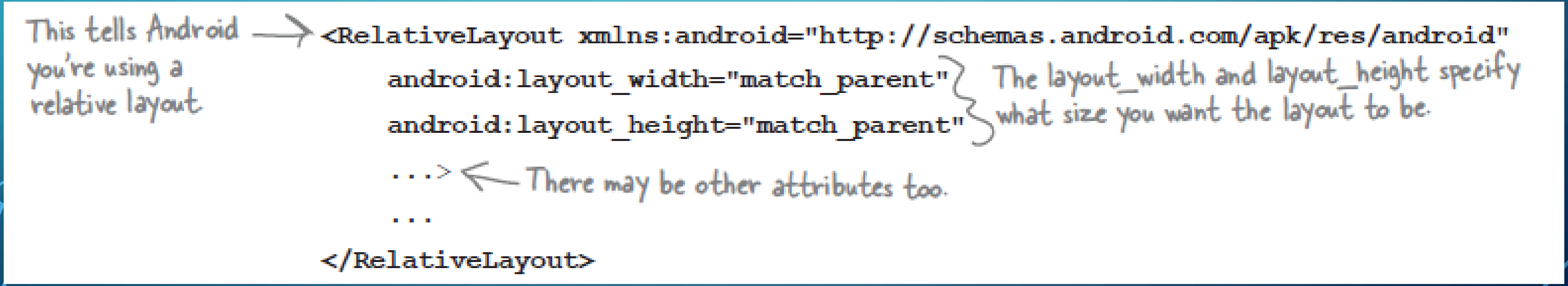
- A **relative layout** displays its views in relative positions. You define the position of each view relative to other views in the layout, or relative to its parent layout.

- As an example, you can choose to position a text view relative to the top of the parent layout, a spinner underneath the text view, and a button relative to the bottom of the parent layout.


Views can be placed relative to the parent layout...
...or to other views

RELATIVE LAYOUT

- As you already know, a relative layout allows you to position views relative to the parent layout, or relative to other views in the layout.
- You define a relative layout using the <RelativeLayout> element like this:



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

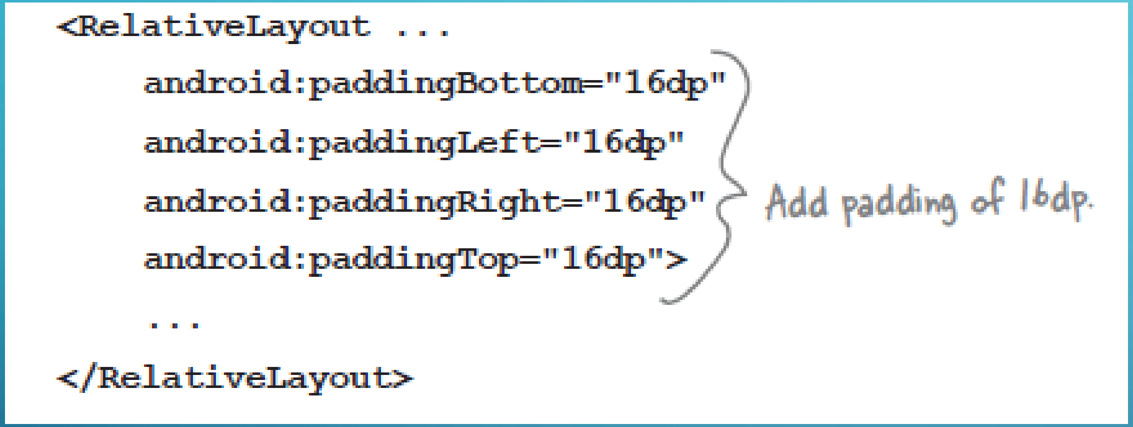
This tells Android you're using a relative layout
The layout_width and layout_height specify what size you want the layout to be.
There may be other attributes too.

LAYOUT WIDTH AND HEIGHT

- The android:layout_width and android:layout_height attributes specify how wide and high you want the layout to be. **These attributes are mandatory for all types of layout and view.**
- You can set android:layout_width and android:layout_height to "match_parent", "wrap_content" or a specific size such as 10dp - 10 density-independent pixels.
- "wrap_content" means that you want the layout to be just big enough to hold all of the views inside it,
- and "match_parent" means that you want the layout to be as big as its parent—in this case, as big as the device screen minus any padding. You will usually set the layout width and height to "match_parent".
- You may sometimes see android:layout_width and android:layout_height set to "fill_parent". "fill_parent" was used in older versions of Android, and it's now replaced by "match_parent". "fill_parent" is deprecated.

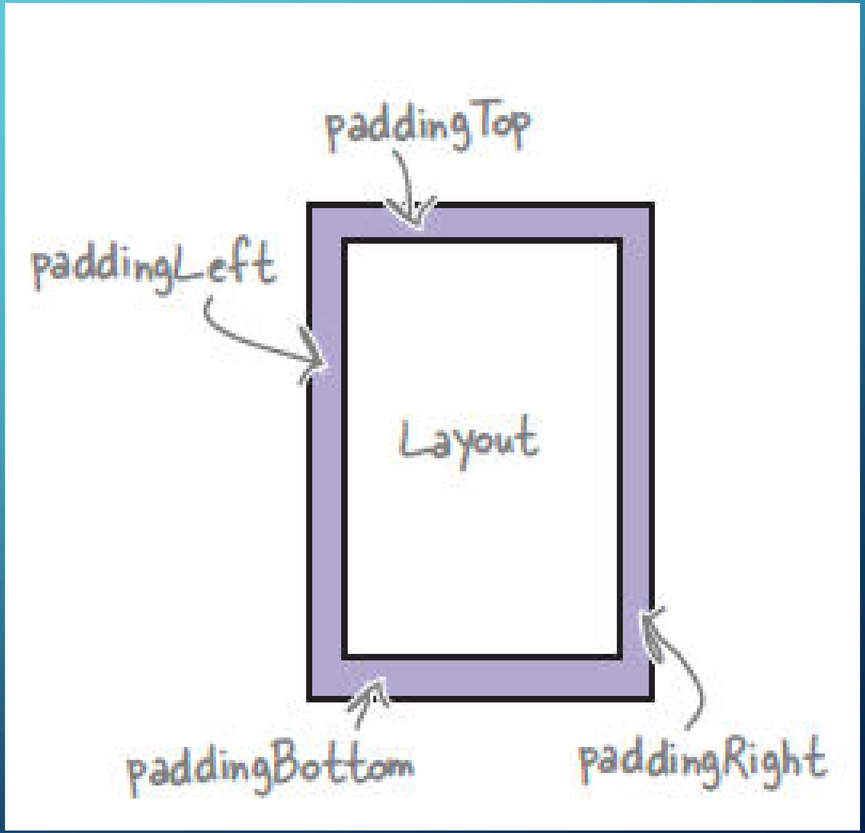
ADDING PADDING

- If you want there to be a bit of space around the edge of the layout, you can set padding attributes.



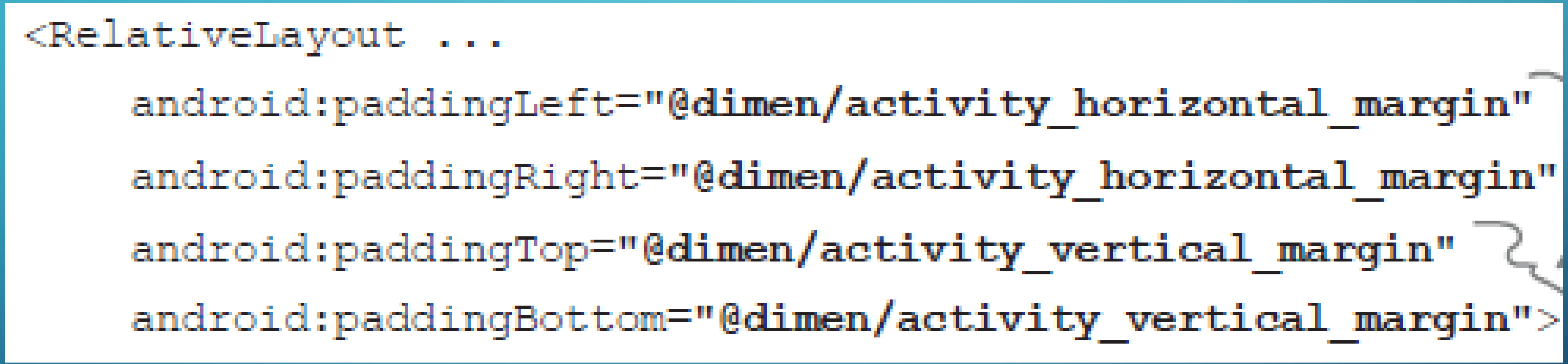
```
<RelativeLayout ...
    android:paddingBottom="16dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp">
    ...
</RelativeLayout>
```

Add padding of 16dp.



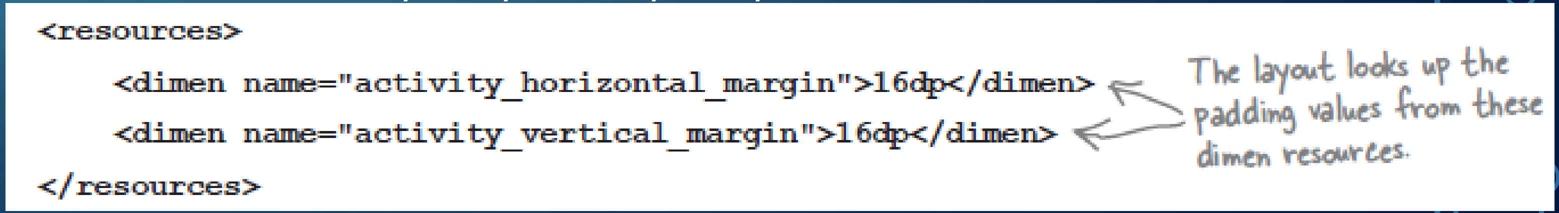
ADDING PADDING - ALTERNATIVE

- An alternative approach is to specify the padding in a dimension resource file instead. Using a dimension resource file makes it easier to maintain the padding of all the layouts in your app.



```
<RelativeLayout ...
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin">
```

- Android then looks up the values of the attributes at runtime in the dimension resource file. This file is located in the *app/src/main/res/values* folder, and it's usually called *dimens.xml*:

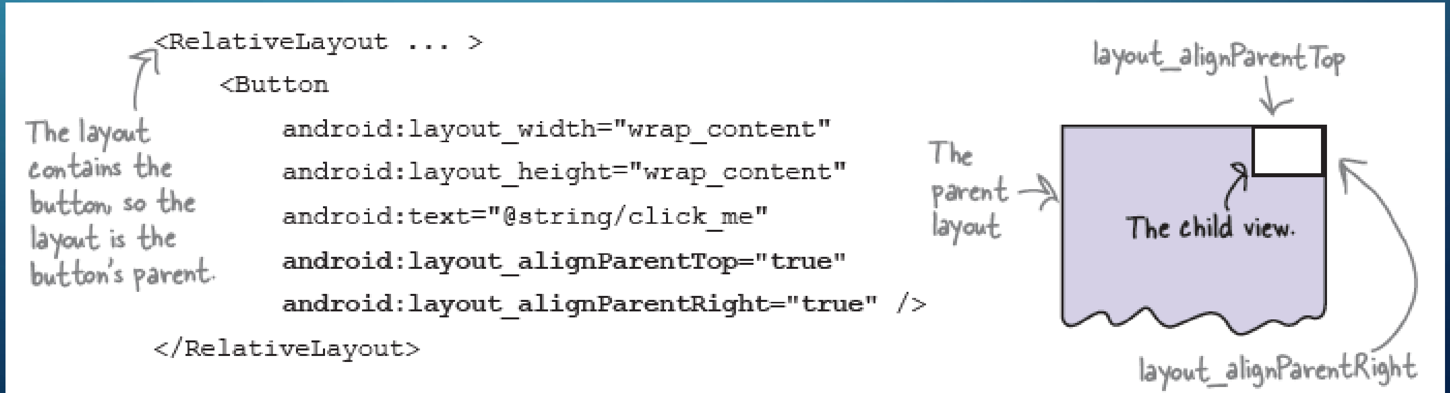


```
<resources>
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

The layout looks up the padding values from these dimen resources.

POSITIONING VIEWS RELATIVE TO THE PARENT LAYOUT

- When you use a relative layout, you need to tell Android where you want its views to appear relative to other views in the layout, or to its parent. A view's parent is the layout that contains the view.
- If you want a view to always appear in a particular position on the screen, irrespective of the screen size or orientation, you need to position the view relative to its *parent*. As an example, here's how you'd make sure a button always appears in the top-right corner of the layout:



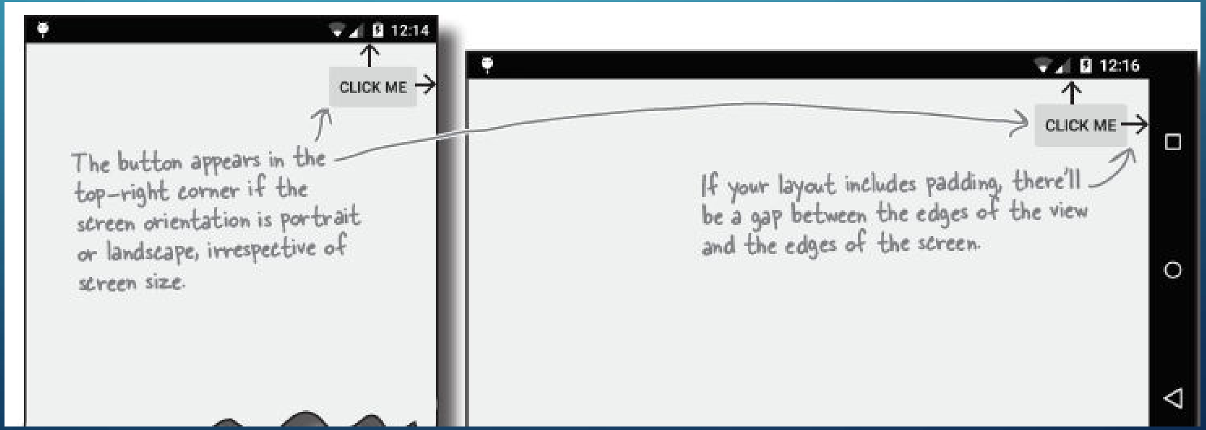
```
<RelativeLayout ... >
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/click_me"
        android:layout_alignParentTop="true"
        android:layout_alignParentRight="true" />
</RelativeLayout>
```

POSITIONING RELATIVE TO THE PARENT LAYOUT

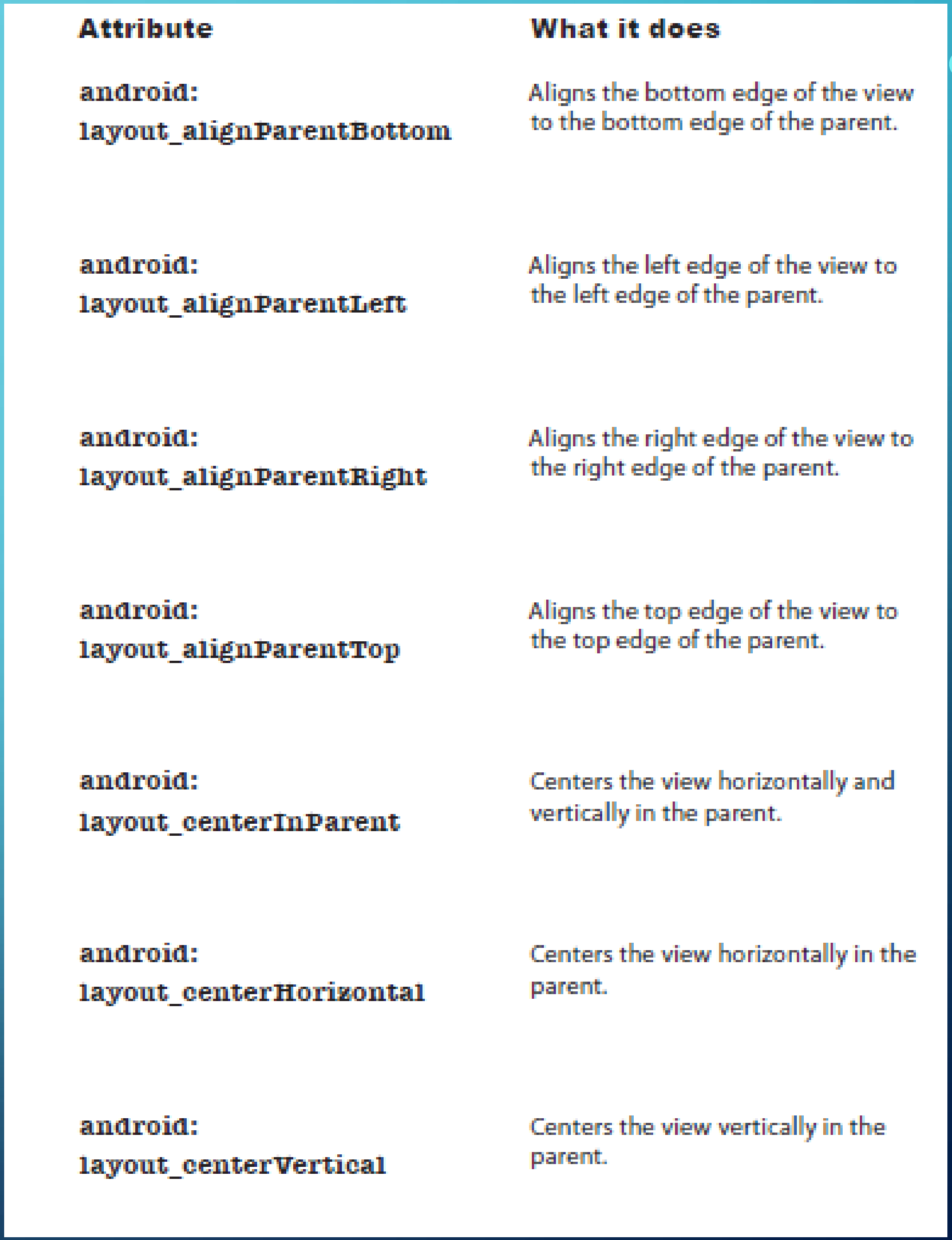
- The lines of code

```
android:layout_alignParentTop="true"
android:layout_alignParentRight="true"
```

mean that the top edge of the button is aligned to the top edge of the layout, and the right edge of the button is aligned to the right edge of the layout. This will be the case no matter what the screen size or orientation of your device:


12:14
CLICK ME
The button appears in the top-right corner if the screen orientation is portrait or landscape, irrespective of screen size.
12:16
CLICK ME
If your layout includes padding, there'll be a gap between the edges of the view and the edges of the screen.

ATTRIBUTE FOR POSITIONING RELATIVE TO PARENT LAYOUT



Attribute	What it does
android: layout_alignParentBottom	Aligns the bottom edge of the view to the bottom edge of the parent.
android: layout_alignParentLeft	Aligns the left edge of the view to the left edge of the parent.
android: layout_alignParentRight	Aligns the right edge of the view to the right edge of the parent.
android: layout_alignParentTop	Aligns the top edge of the view to the top edge of the parent.
android: layout_centerInParent	Centers the view horizontally and vertically in the parent.
android: layout_centerHorizontal	Centers the view horizontally in the parent.
android: layout_centerVertical	Centers the view vertically in the parent.

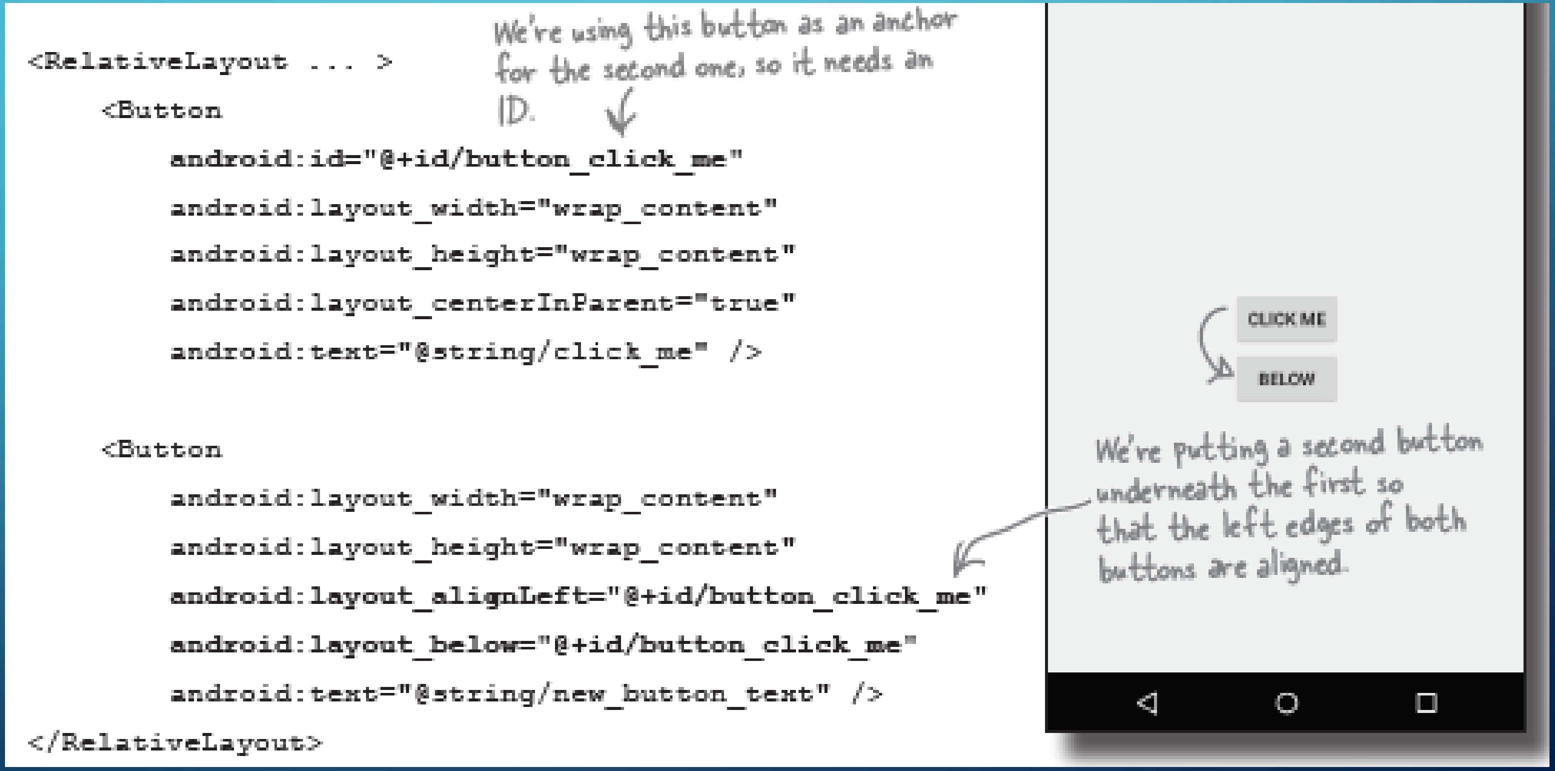
POSITIONING VIEWS RELATIVE TO OTHER VIEWS

- In addition to positioning views relative to the parent layout, you can also position views relative to other views. You do this when you want views to stay aligned in some way, irrespective of the screen size or orientation.
- In order to position a view relative to another view, the view you're using as an anchor must be given an ID using the android:id

  android:id="@+id/button_click_me"

- The syntax "@+id" tells Android to include the ID as a resource in its resource file *R.java*. If you miss out the "+", Android won't add the ID as a resource and you'll get errors in your code.

- Here’s how you create a layout with two buttons, with one button centered in the middle of the layout, and the second button positioned underneath the first:

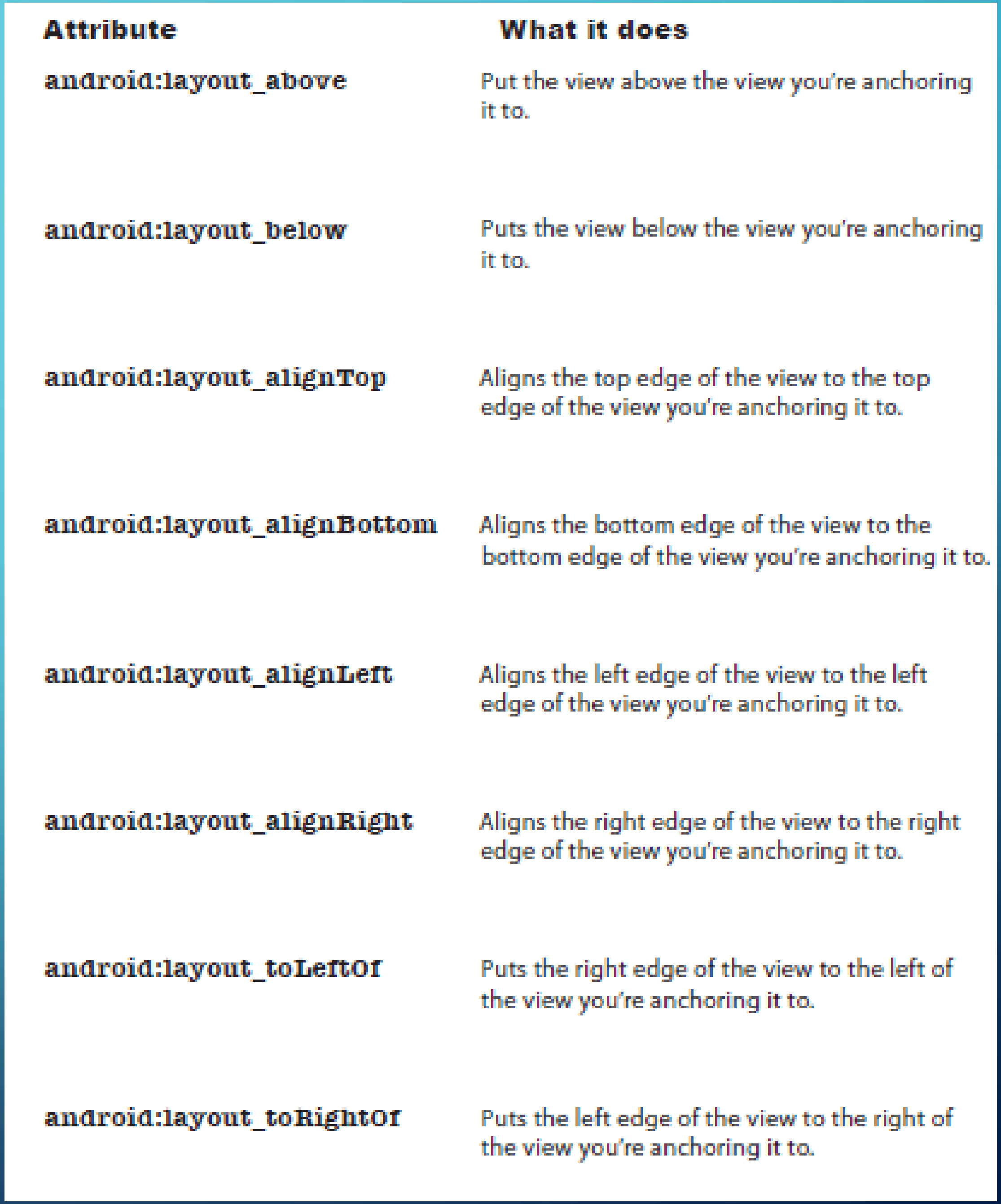


```
<RelativeLayout ... >
    <Button
        android:id="@+id/button_click_me"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="@string/click_me" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignLeft="@+id/button_click_me"
        android:layout_below="@+id/button_click_me"
        android:text="@string/new_button_text" />
</RelativeLayout>
```

We're using this button as an anchor for the second one, so it needs an ID.

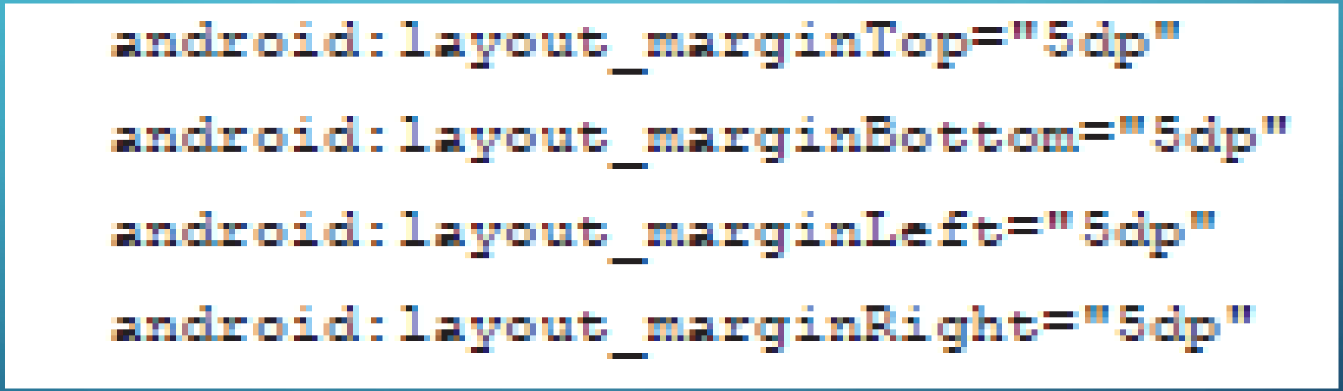
We're putting a second button underneath the first so that the left edges of both buttons are aligned.



Attribute	What it does
android:layout_above	Put the view above the view you're anchoring it to.
android:layout_below	Puts the view below the view you're anchoring it to.
android:layout_alignTop	Aligns the top edge of the view to the top edge of the view you're anchoring it to.
android:layout_alignBottom	Aligns the bottom edge of the view to the bottom edge of the view you're anchoring it to.
android:layout_alignLeft	Aligns the left edge of the view to the left edge of the view you're anchoring it to.
android:layout_alignRight	Aligns the right edge of the view to the right edge of the view you're anchoring it to.
android:layout_toLeftOf	Puts the right edge of the view to the left of the view you're anchoring it to.
android:layout_toRightOf	Puts the left edge of the view to the right of the view you're anchoring it to.

MARGIN BETWEEN VIEWS

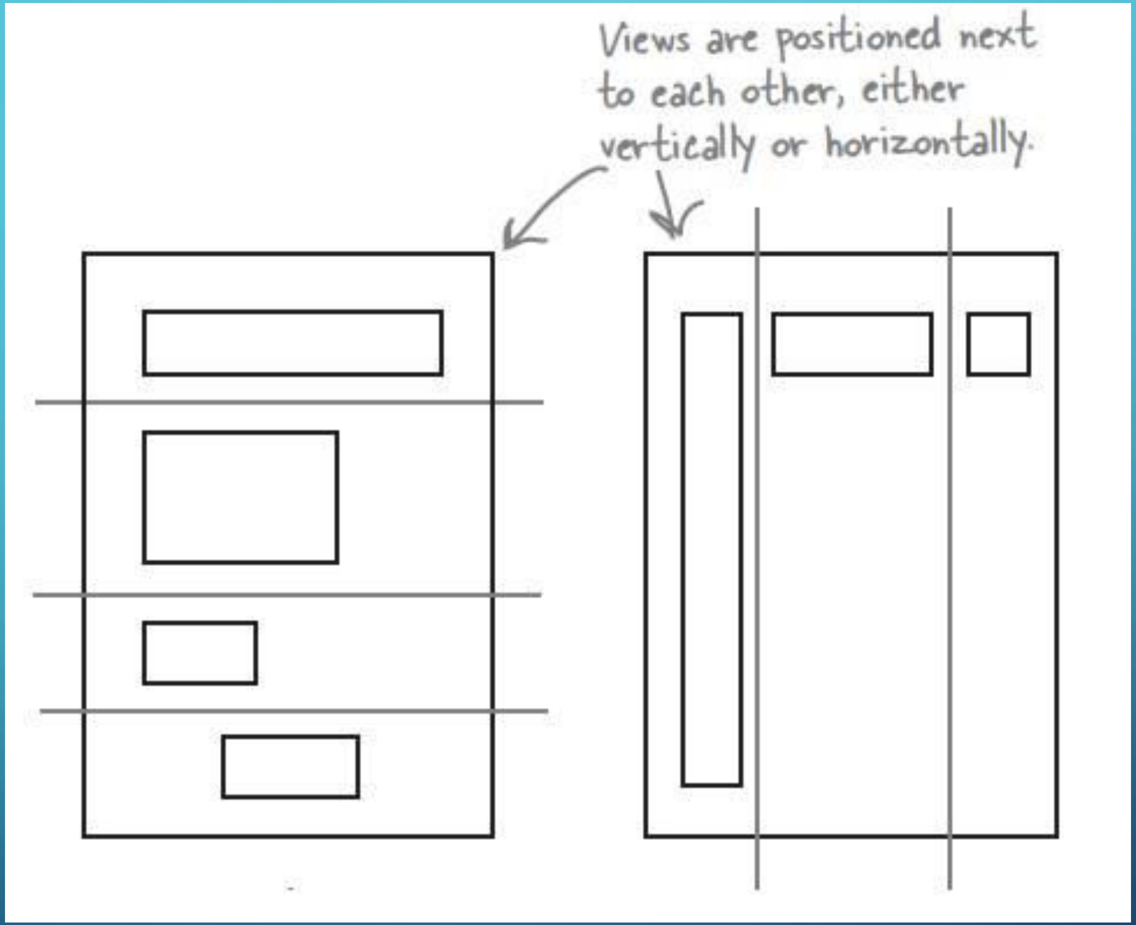
- When you use any of the layout attributes to position a view, the layout doesn't leave much of a gap. You can increase the size of the gap between views by adding one or more margins to the view.



```
android:layout_marginTop="5dp"
android:layout_marginBottom="5dp"
android:layout_marginLeft="5dp"
android:layout_marginRight="5dp"
```

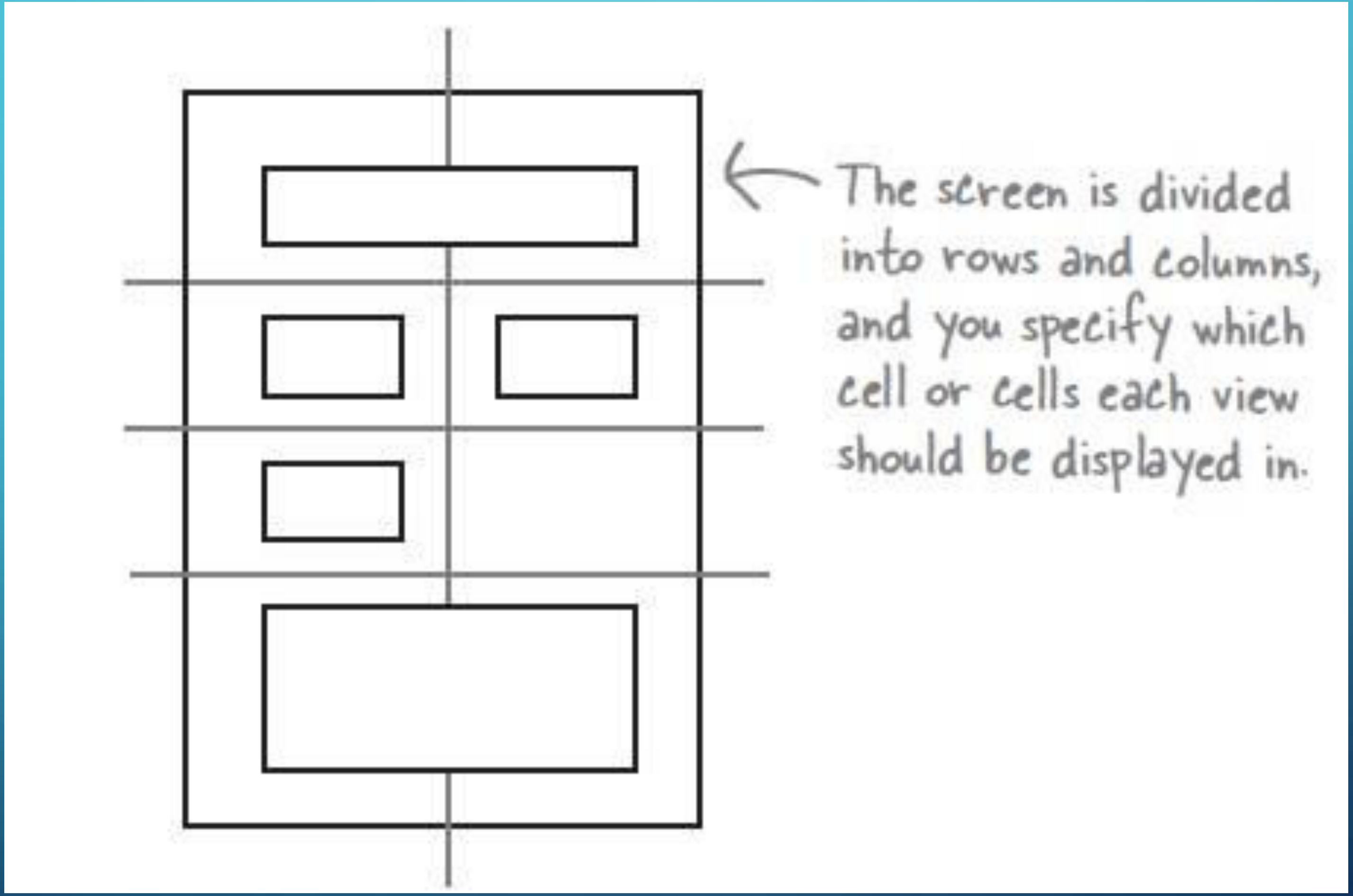
LINEARLAYOUT

- A **linear layout** displays views next to each other either vertically or horizontally. If it's vertically, the views are displayed in a single column. If it's horizontally, the views are displayed in a single row.


Views are positioned next to each other, either vertically or horizontally.

GRID LAYOUT

- A **grid layout** divides the screen into a grid of rows, columns, and cells. You specify how many columns your layout should have, where you want your views to appear, and how many rows or columns they should span.

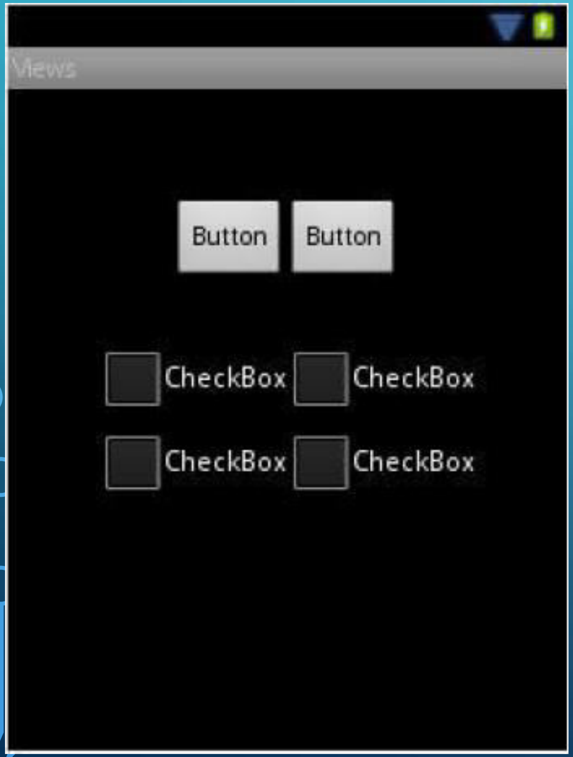

The screen is divided into rows and columns, and you specify which cell or cells each view should be displayed in.

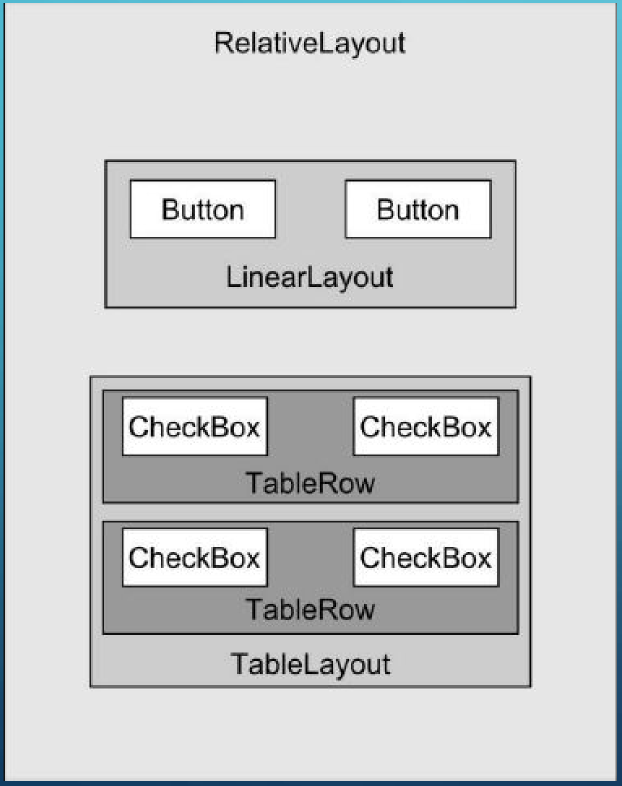
TABLELAYOUT, FRAMELAYOUT, ABSOLUTELAYOUT

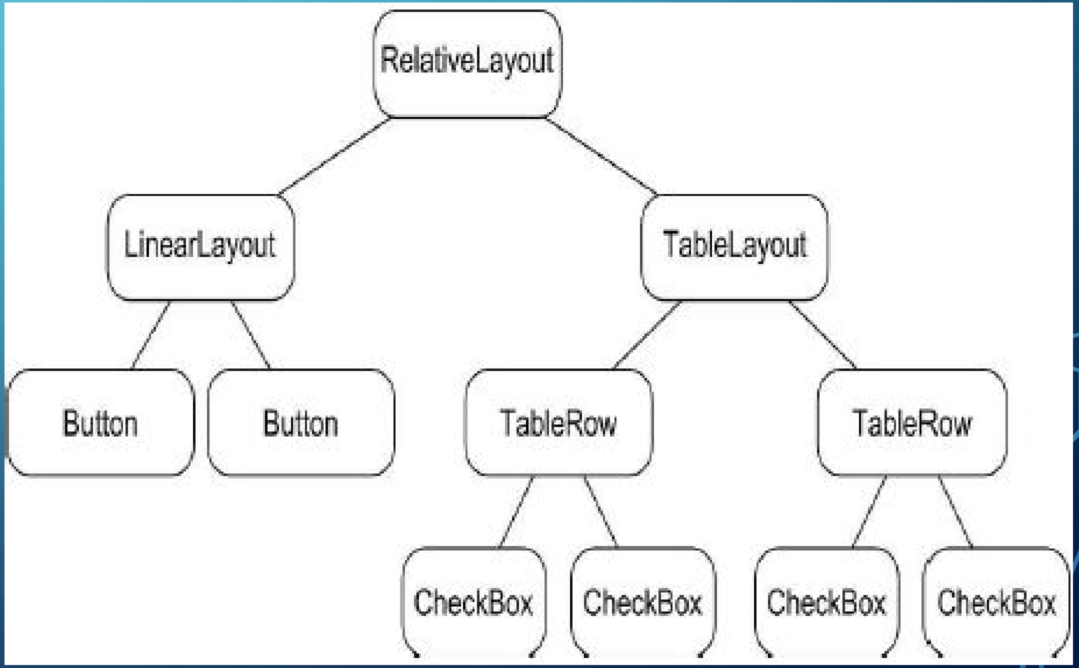
- **TableLayout** – Arranges child views into a grid format of rows and columns. Each row within a table is represented by a *TableRow* object child, which, in turn, contains a view object for each cell.
- **FrameLayout** – The purpose of the FrameLayout is to allocate an area of screen, typically for the purposes of displaying a single view. If multiple child views are added they will, by default, appear on top of each other positioned in the top left hand corner of the layout area. Alternate positioning of individual child views can be achieved by setting gravity values on each child. For example, setting a *center_vertical* gravity on a child will cause it to be positioned in the vertical center of the containing FrameLayout view.
- **AbsoluteLayout** – Allows child views to be positioned at specific X and Y coordinates within the containing layout view. Use of this layout is discouraged since it lacks the flexibility to respond to changes in screen size and orientation.

VIEW HIERARCHY

- Each view in a user interface represents a rectangular area of the display. A view is responsible for what is drawn in that rectangle and for responding to events that occur within that part of the screen (such as a touch event).







VIEW HIERARCHY

- A user interface screen is comprised of a view hierarchy with a root view positioned at the top of the tree and child views positioned on branches below. The child of a container view appears on top of its parent view and is constrained to appear within the bounds of the parent view's display area.
- In addition to the visible button and checkbox views, the user interface actually includes a number of layout views that control how the visible views are positioned.