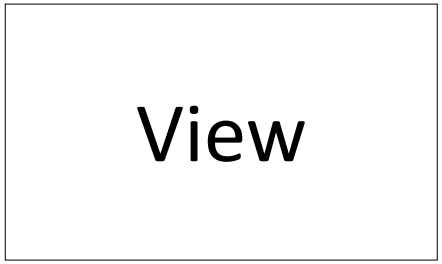
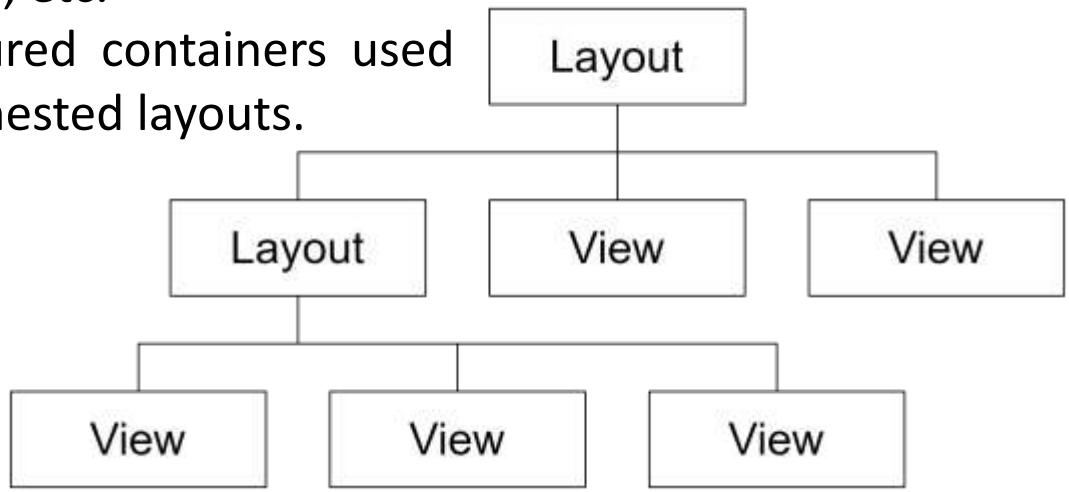
Graphical User Interfaces
Basic widgets

The View class

- The **View class** is the Android's most basic component from which user interfaces can be created. It acts as a container of displayable elements.
- A **View** occupies a rectangular area on the screen and is responsible for *drawing* and *event handling*.
- **Widgets** are subclasses of View. They are used to create interactive UI components such as buttons, checkboxes, labels, text fields, etc.
- **Layouts** are invisible structured containers used for holding other Views and nested layouts.





Using XML to represent UIs

Actual UI displayed by the app

Text version: *activity_main.xml* file ⟶

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

    <EditText
        android:id="@+id/editText1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="36dp"
        android:text="@string/edit_user_name"
        android:ems="12" >
        <requestFocus />
    </EditText>

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/editText1"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="48dp"
        android:text="@string/btn_go"  />

</RelativeLayout>
```

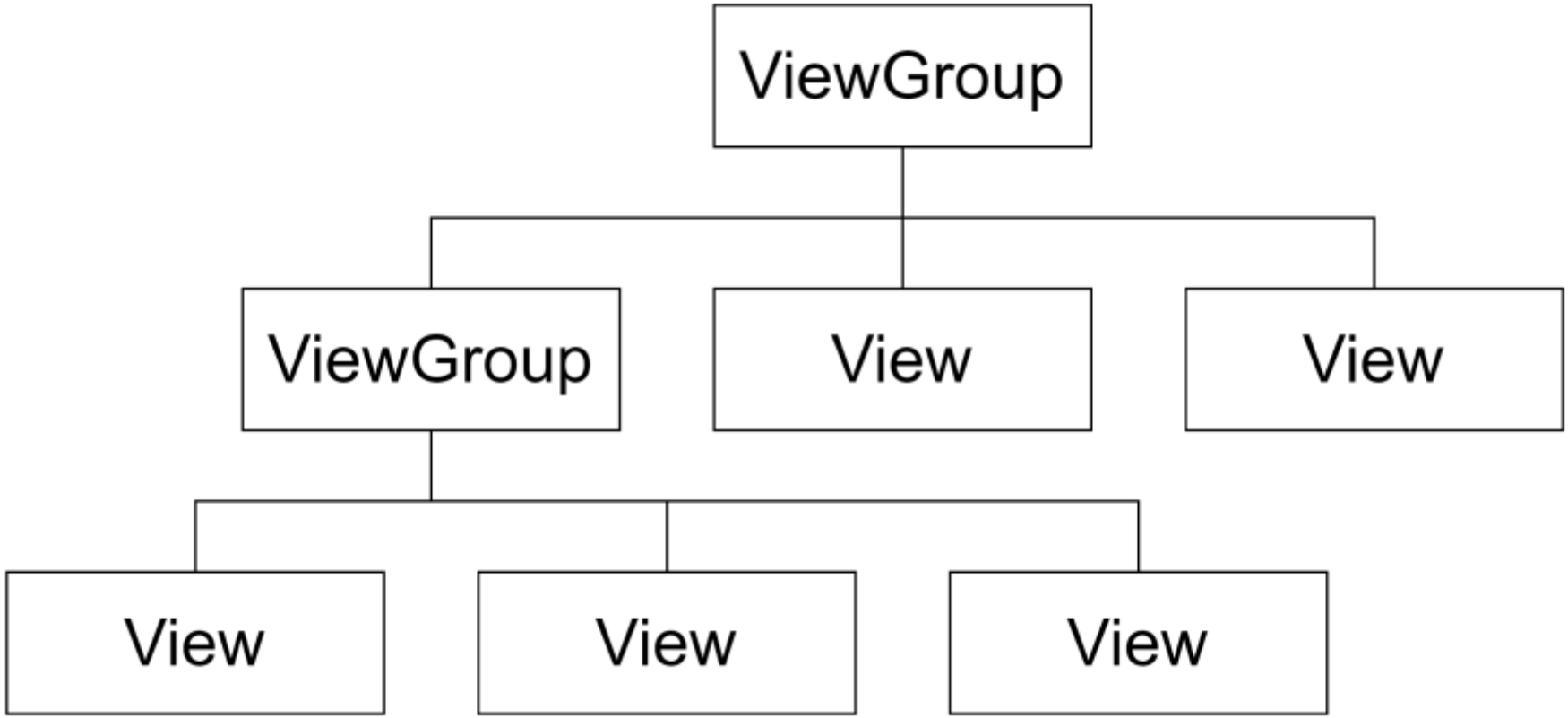
Nesting XML Layouts

- An Android's **XML view** file consists of a **layout** design holding a hierarchical arrangement of its contained elements.
- The inner elements could be basic widgets or user-defined nested layouts holding their own viewgroups.
- An Activity uses the `setContentView(R.layout.xmlfilename)` method to render a view on the device's screen.

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


</LinearLayout>
```

Widgets and other nested layouts

Nesting XML Layouts

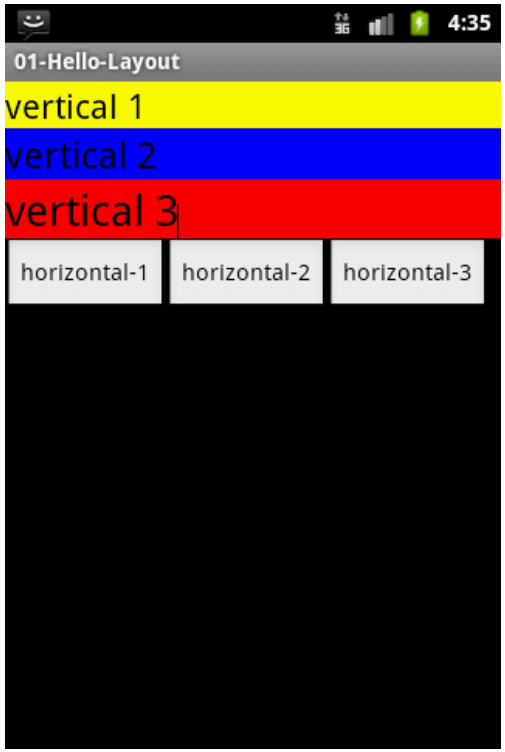


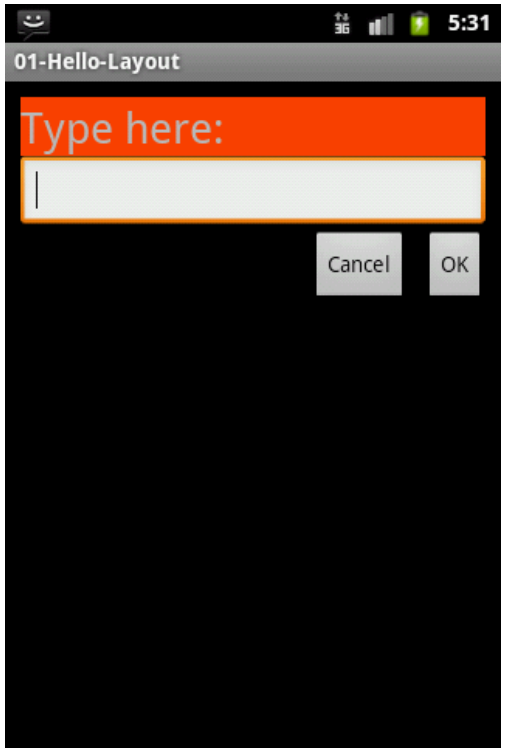
Setting Views to Work

Dealing with widgets & layouts typically involves the following operations:

1. **Set properties:** For instance, when working with a *TextView* you set the background color, text, font, alignment, size, padding, marging, etc.
2. **Set up listeners:** For example, an image could be programmed to respond to various events such as: click, long-tap, mouse-over, etc.

A Sample of Common Android LAYOUTS





Linear Layout
A LinearLayout places its inner views either in horizontal or vertical disposition.

Relative Layout
A RelativeLayout is a ViewGroup that allows you to position elements relative to each other.

Table Layout
A TableLayout is a ViewGroup that places elements using a row & column disposition.

A Sample of Common Android WIDGETS

GalleryView

TabWidget

Spinner

TimePicker
AnalogClock
DatePicker

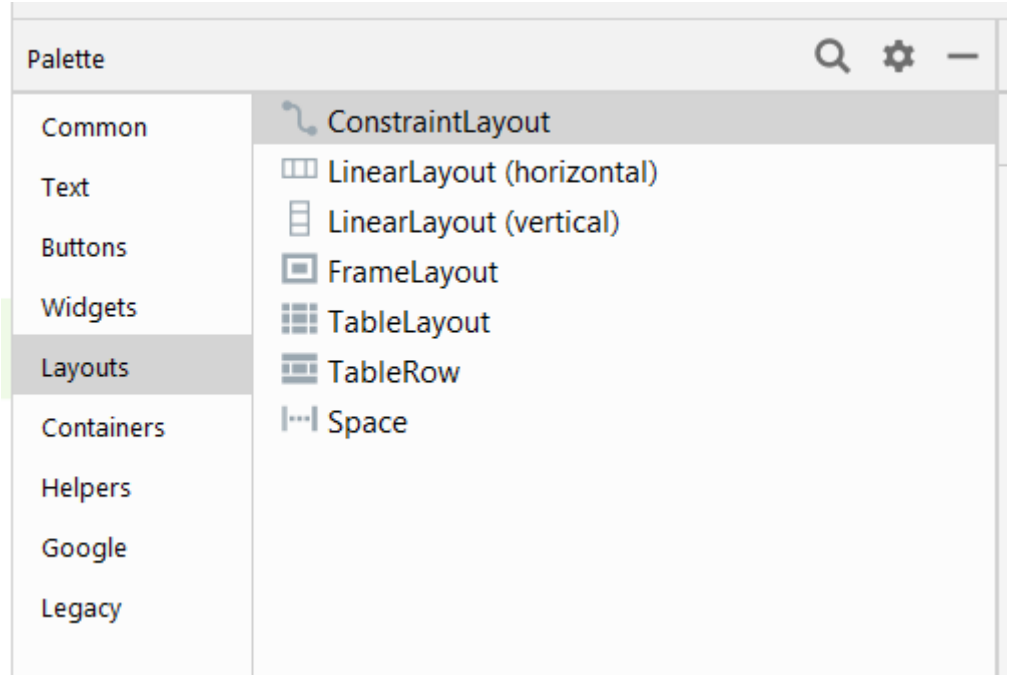
A *DatePicker* is a widget that allows the user to select a month, day and year.

Form Controls

Includes a variety of typical form widgets, like: *buttons, image buttons, text fields, checkboxes* and *radio buttons*.

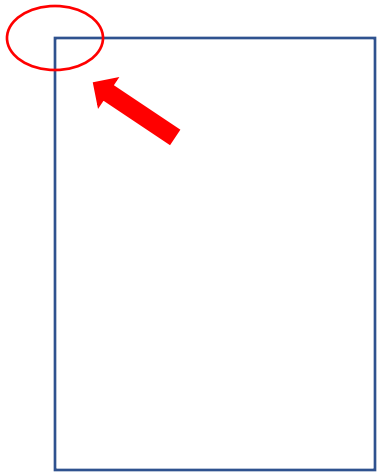
GUI Elements: The LAYOUT

- Android GUI *Layouts* are containers having a predefined structure and placement policy such as relative, linear horizontal, grid-like, etc.
- **Layouts can be nested**, therefore a cell, row, or column of a given layout could be another layout.



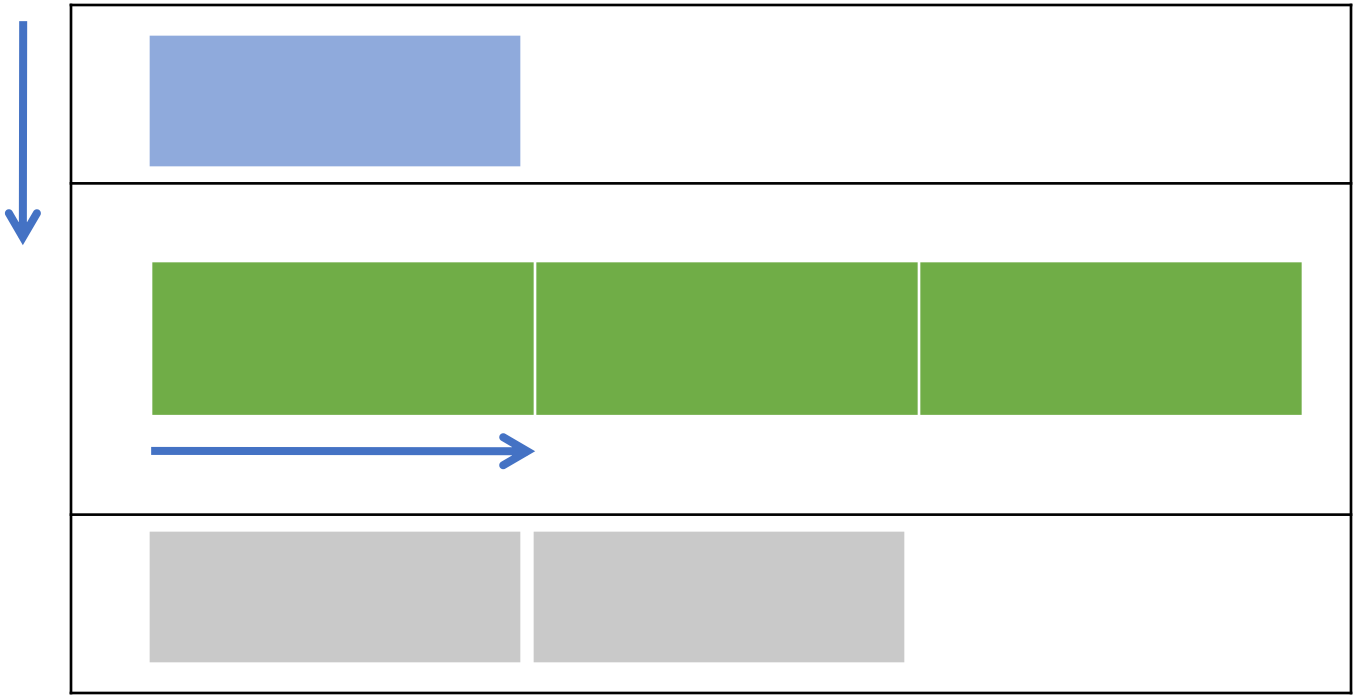
FrameLayout

- The **FrameLayout** is the simplest type of GUI container.
- It is useful as an *outermost* container holding a window.
- Allows you to define how much of the screen (high, width) is to be used.
- All its children elements are *aligned to the top left corner of the screen.*



LinearLayout

- The **LinearLayout** supports a filling strategy in which new elements are stacked either in a horizontal or vertical fashion.
- If the layout has a vertical orientation new *rows* are placed one on top of the other.
- A horizontal layout uses a side-by-side *column* placement policy.



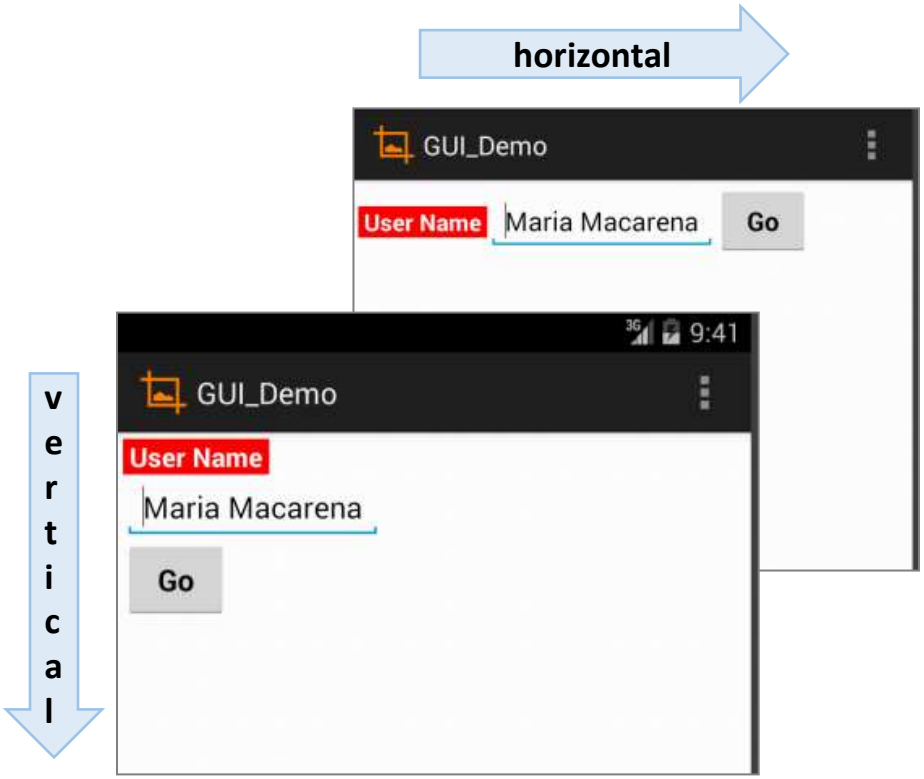
Setting Attributes

Configuring a **LinearLayout** usually requires you to set the following attributes:

- **orientation** *(vertical, horizontal)*
- **fill model** *(match_parent, wrap_contents)*
- **weight** *(0, 1, 2, …n)*
- **gravity** *(top, bottom, center,…)*
- **padding** *(dp – dev. independent pixels)*
- **margin** *(dp – dev. independent pixels)*

LinearLayout : Orientation

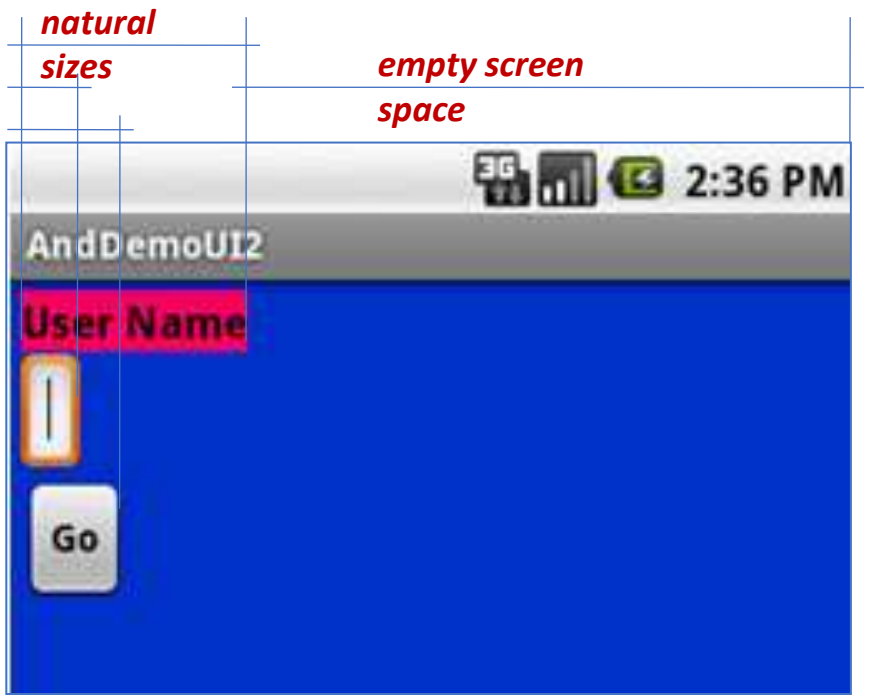
The **`android:orientation`** property can be set to: **horizontal** for columns, or **vertical** for rows. Use *`setOrientation()`* for runtime changes.




```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/and
roid"
    android:id="@+id/myLinearLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    android:padding="4dp" >

    <TextView
        android:id="@+id/labelUserName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="#ffff0000"
        android:text=" User Name "
        android:textColor="#ffffffff"
        android:textSize="16sp"
        android:textStyle="bold" />

    <EditText
        android:id="@+id/ediName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Maria Macarena"
        android:textSize="18sp" />

    <Button
        android:id="@+id/btnGo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Go"
        android:textStyle="bold" />
</LinearLayout>
```

LinearLayout : Fill Model

- Widgets have a "natural size“ based on their included text (*rubber band* effect).
- On occasions you may want your widget to have a specific space allocation (height, width) even if no text is initially provided (as is the case of the empty text box shown below).



LinearLayout : Fill Model

All widgets inside a LinearLayout **must** include 'width' and 'height' attributes.

```
android:layout_width
android:layout_height
```

Values used in defining height and width can be:

1. A specific dimension such as **125dp** (device independent pixels **dip**)

2. **wrap_content** indicates the widget should just fill up its natural space.

3. **match_parent** (previously called '**fill_parent**') indicates the widget wants to be as big as the enclosing parent.

LinearLayout : Fill Model

Medium resolution is: 320 x 480 dpi.
Shown on a Gingerbread device

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/myLinearLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#ff0033cc"
    android:orientation="vertical"
    android:padding="6dp" >

    <TextView
        android:id="@+id/labelUserName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#ffff0066"
        android:text="User Name"
        android:textColor="#ff000000"
        android:textSize="16sp"
        android:textStyle="bold" />

    <EditText
        android:id="@+id/ediName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="18sp" />

    <Button
        android:id="@+id/btnGo"
        android:layout_width="125dp"
        android:layout_height="wrap_content"
        android:text="Go"
        android:textStyle="bold" />
</LinearLayout>
```

Row-wise

Use all the row

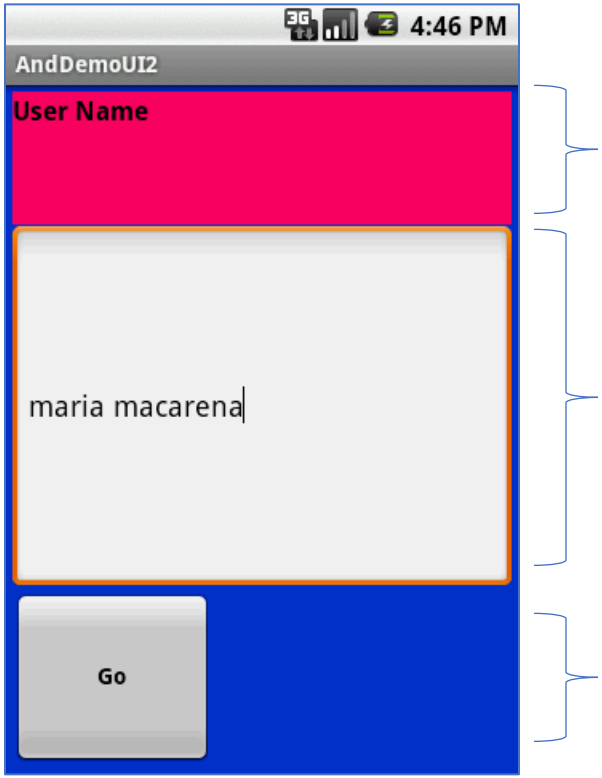
Specific size: 125dp

LinearLayout : Weight

The extra space left unclaimed in a layout could be assigned to any of its inner components by setting its **Weight** attribute.
Use **0** if the view should not be stretched. The bigger the weight the larger the extra space given to that widget.

Example
The XML specification for this window is similar to the previous example.

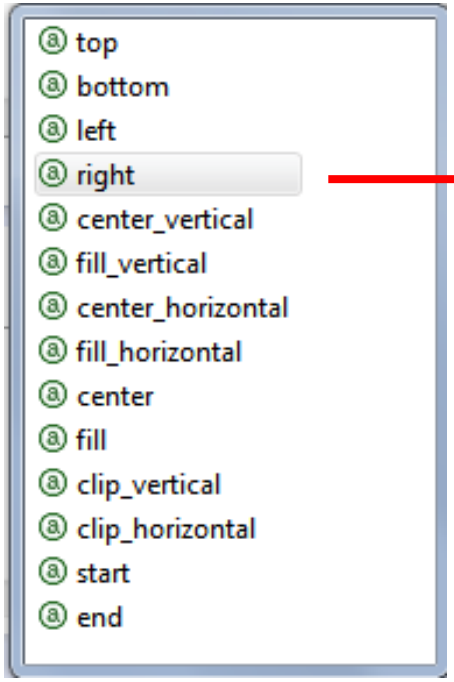
The TextView and Button controls have the additional property
`android:layout_weight="1"`
whereas the EditText control has
`android:layout_weight="2"`
Remember, default value is 0

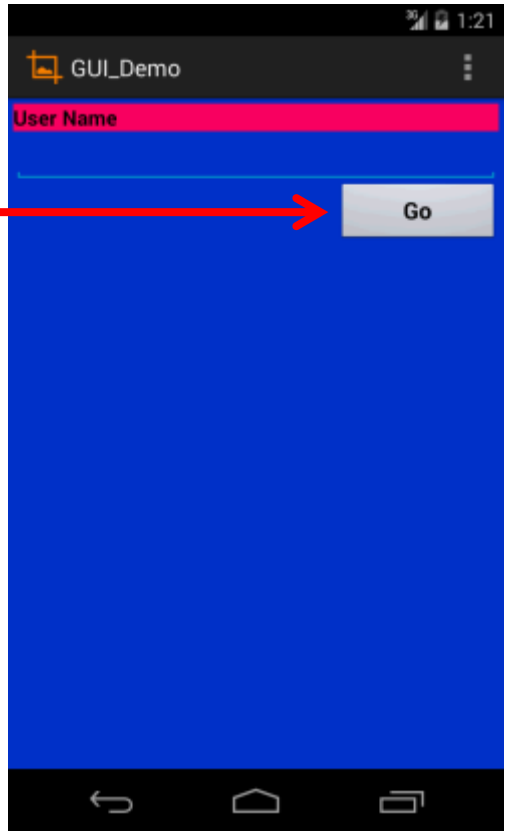


Takes: 2 /(1+1+2) of the screen space

LinearLayout : Gravity

- **Gravity** is used to indicate how a control will align on the screen.
- By default, widgets are ***left-*** and ***top-***aligned.
- You may use the XML property `android:layout_gravity="…"` to set other possible arrangements: ***left, center, right, top, bottom,*** etc.



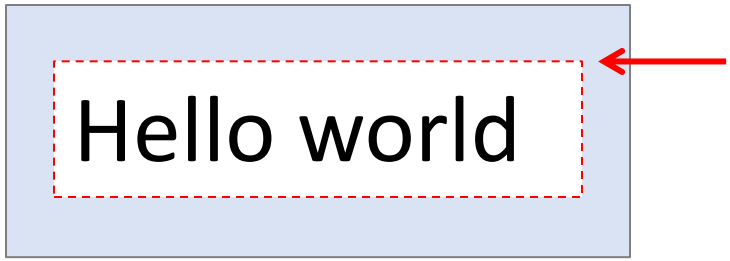


Button has **right** layout_gravity

LinearLayout : Padding

- The **padding** attribute specifies the widget’s internal margin (in **dp** units).

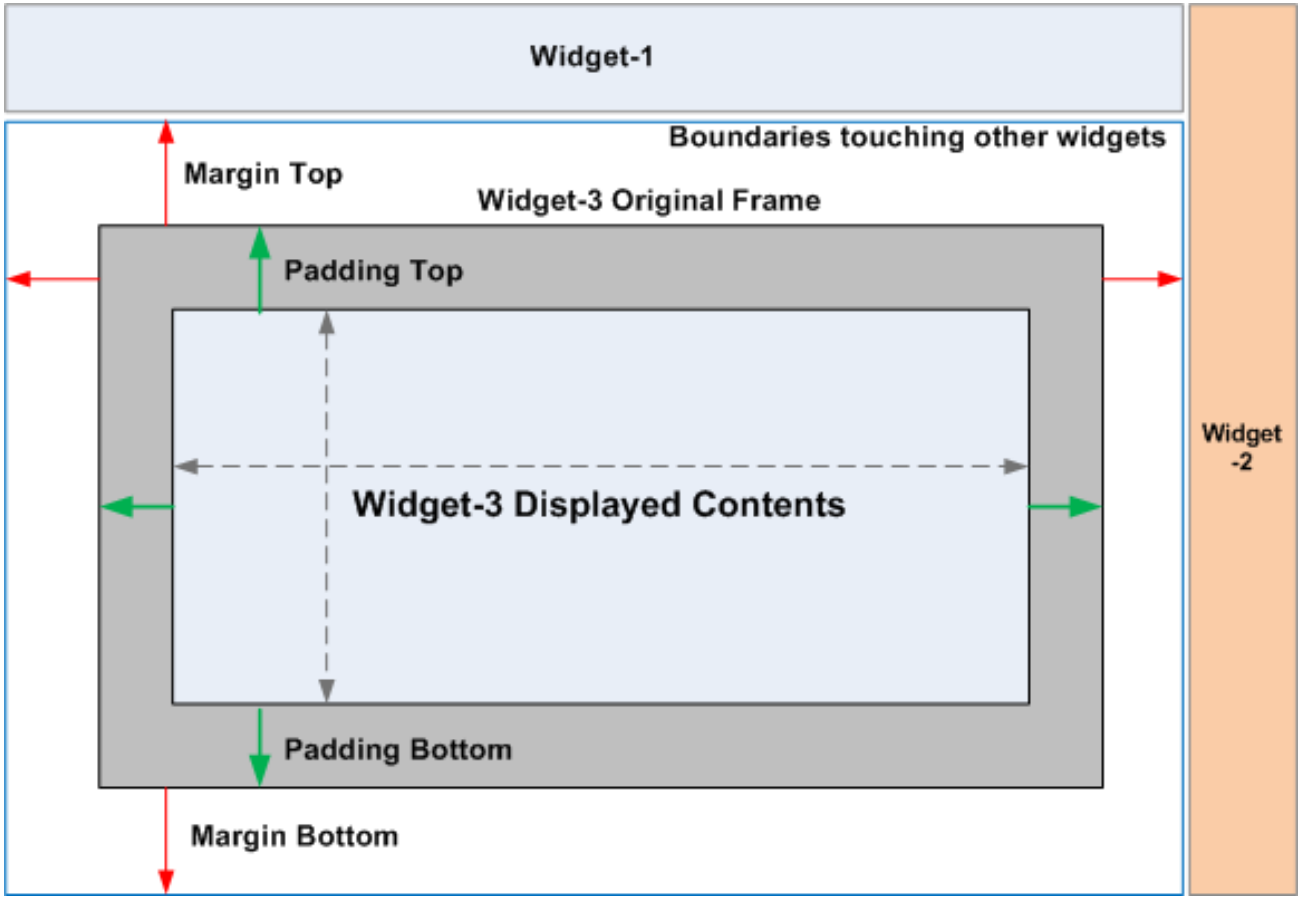
- The internal margin is the extra space between the borders of the widget's "cell" and the actual widget contents.



The ‘blue’ surrounding space around the text represents the inner view’s padding

LinearLayout : Padding and Margin

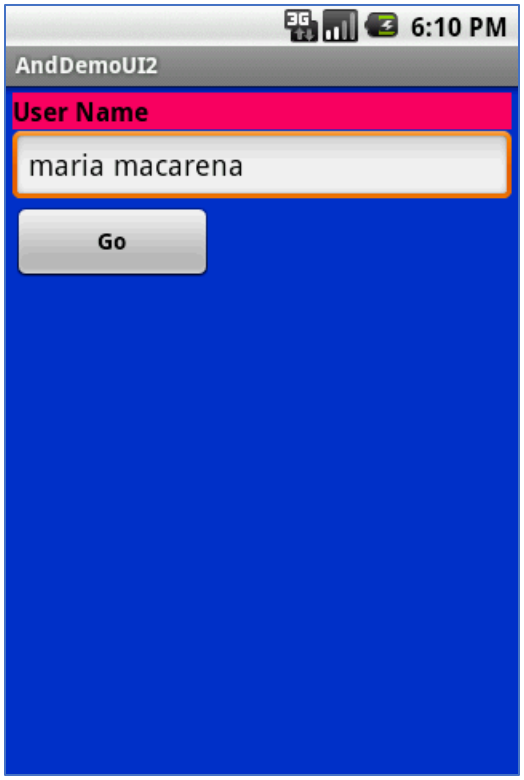
Padding and Margin represent the *internal* and *external* spacing between a widget and its included and surrounding context (respectively).

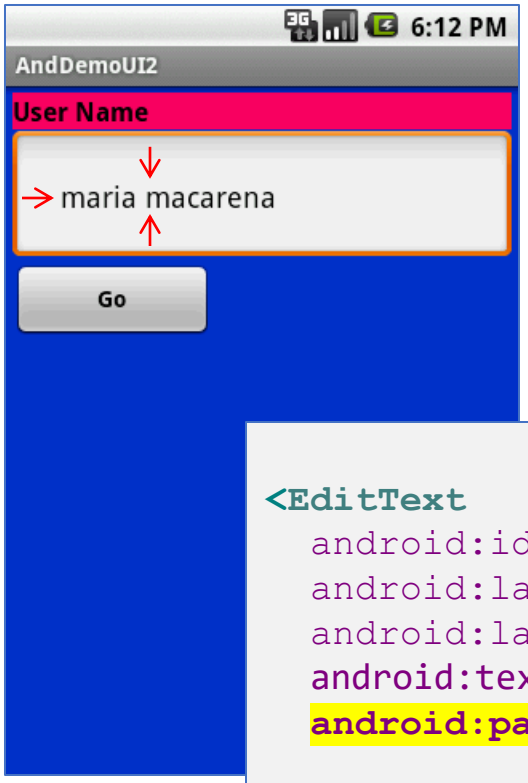


LinearLayout : Set internal margins using padding

Example:

The EditText box has been changed to include 30dp of padding all around

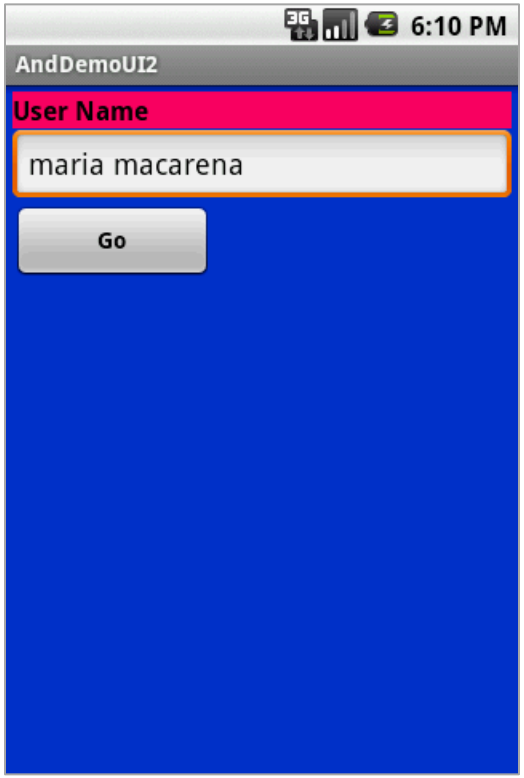




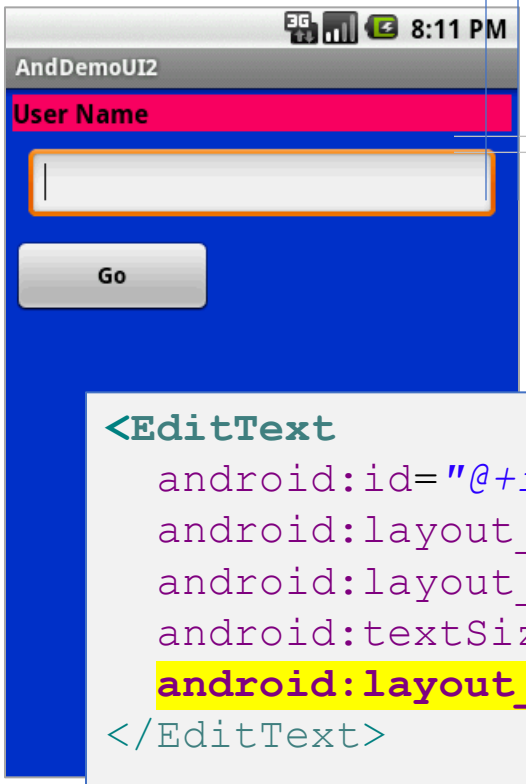

```
<EditText
  android:id="@+id/ediName"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:textSize="18sp"
  android:padding="30dp"  />

...
```

LinearLayout : Set External Margins

- Widgets –by default– are closely displayed next to each other.
- To increase space between them use the **android:layout_margin** attribute



Using default spacing between widgets

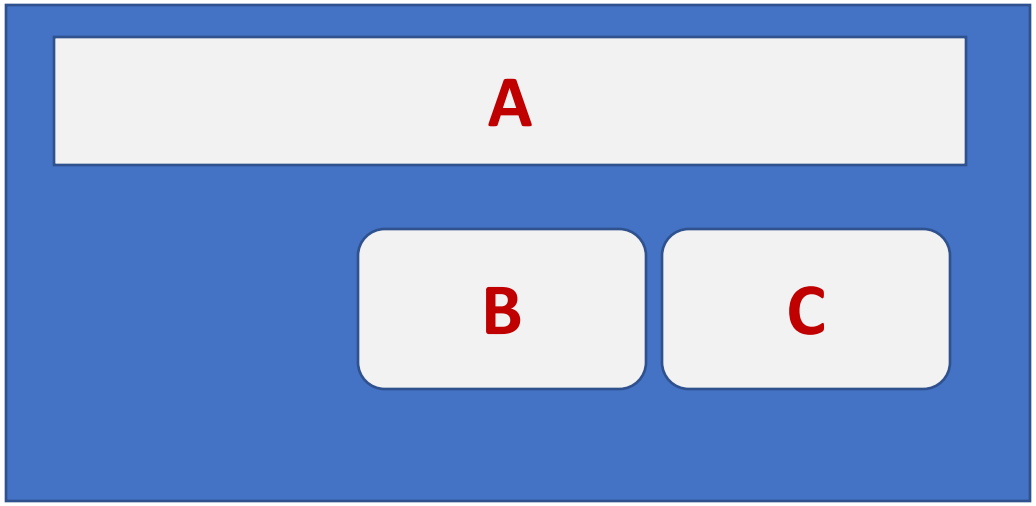


Increased inter-widget space

```
<EditText
  android:id="@+id/ediName"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:textSize="18sp"
  android:layout_margin="6dp">
</EditText>
...
```

Relative Layout

The placement of a widget in a **RelativeLayout** is based on its *positional relationship* to other widgets in the container as well as the parent container.



Example:
A is by the parent's top
C is below A, to its right
B is below A, to the left of C

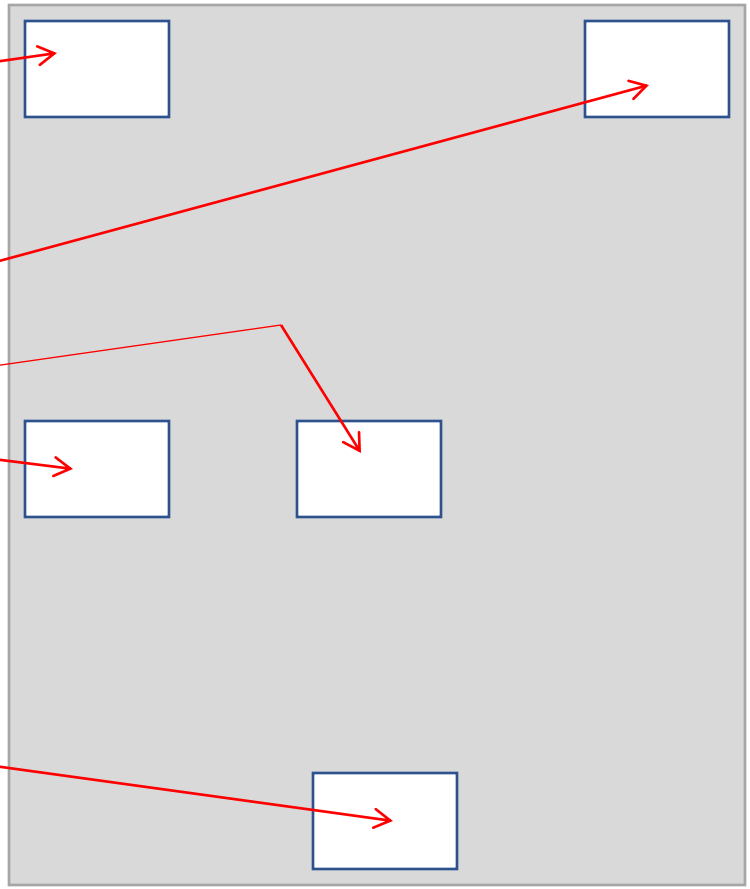
Relative Layout - Example

Location of the button is expressed in reference to its *relative* position with respect to the EditText box.

Relative Layout - Referring to the container

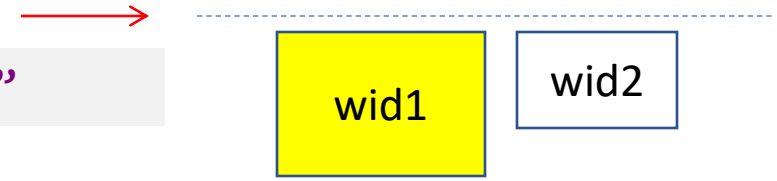
Below there is a sample of various positioning *XML boolean properties* (**true/false**) which are useful for collocating a widget based on the location of its **parent** container.

```
android:layout_alignParentTop
android:layout_alignParentBottom

android:layout_alignParentLeft
android:layout_alignParentRight

android:layout_centerInParent
android:layout_centerVertical
android:layout_centerHorizontal
```



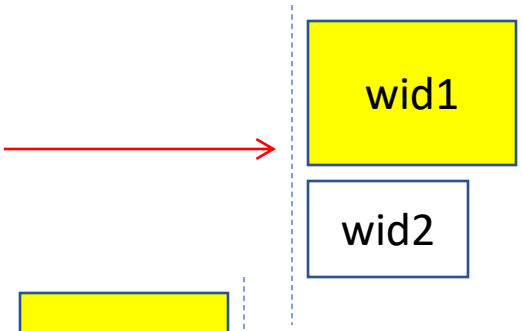
Relative Layout - Referring to Other Widgets

```
android:layout_alignTop="@+id/wid1"
```

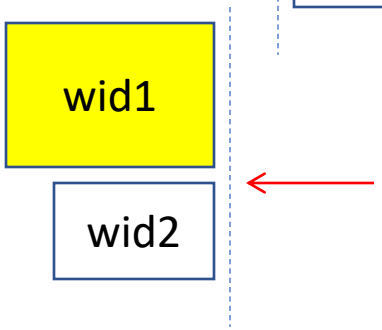



```
android:layout_alignBottom ="@+id/wid1"
```


```
android:layout_alignLeft="@+id/wid1"
```

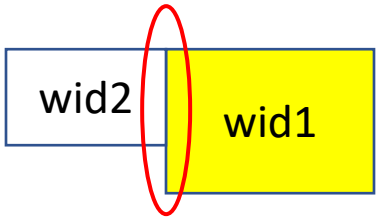



```
android:layout_alignRight="@+id/wid1"
```

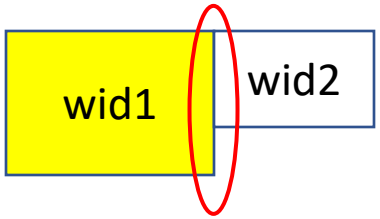


Relative Layout - Referring to Other Widgets

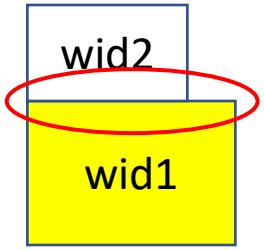
```
android:layout_toLeftOf="@+id/wid1"
```



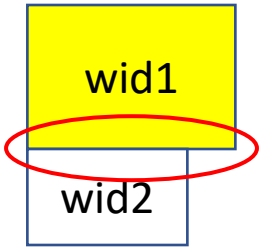

```
android:layout_toRightOf ="@+id/wid1"
```




```
android:layout_above="@+id/wid1"
```




```
android:layout_below="@+id/wid1"
```



Relative Layout - Referring to Other Widgets

When using relative positioning you need to:

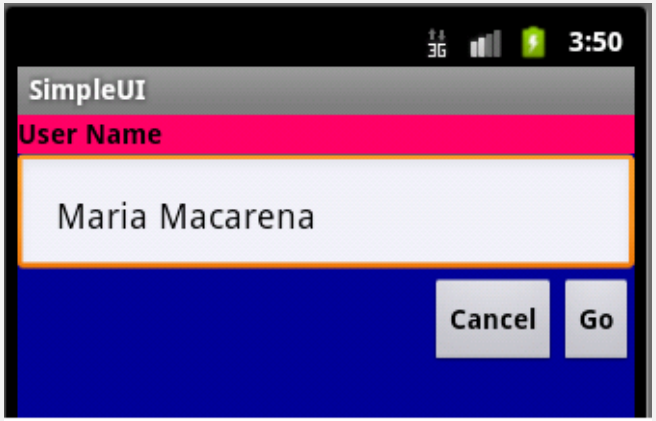
1. Use identifiers (`android:id` attributes) on *all elements* that you will be referring to.

2. XML elements are named using the prefix: `@+id/...` For instance an EditText box could be called: `android:id="@+id/txtUserName"`

3. You must refer only to widgets that have been already defined. For instance a new control to be positioned below the *txtUserName* EditText box could refer to it using:
   `android:layout_below="@+id/txtUserName"`

Relative Layout - Example

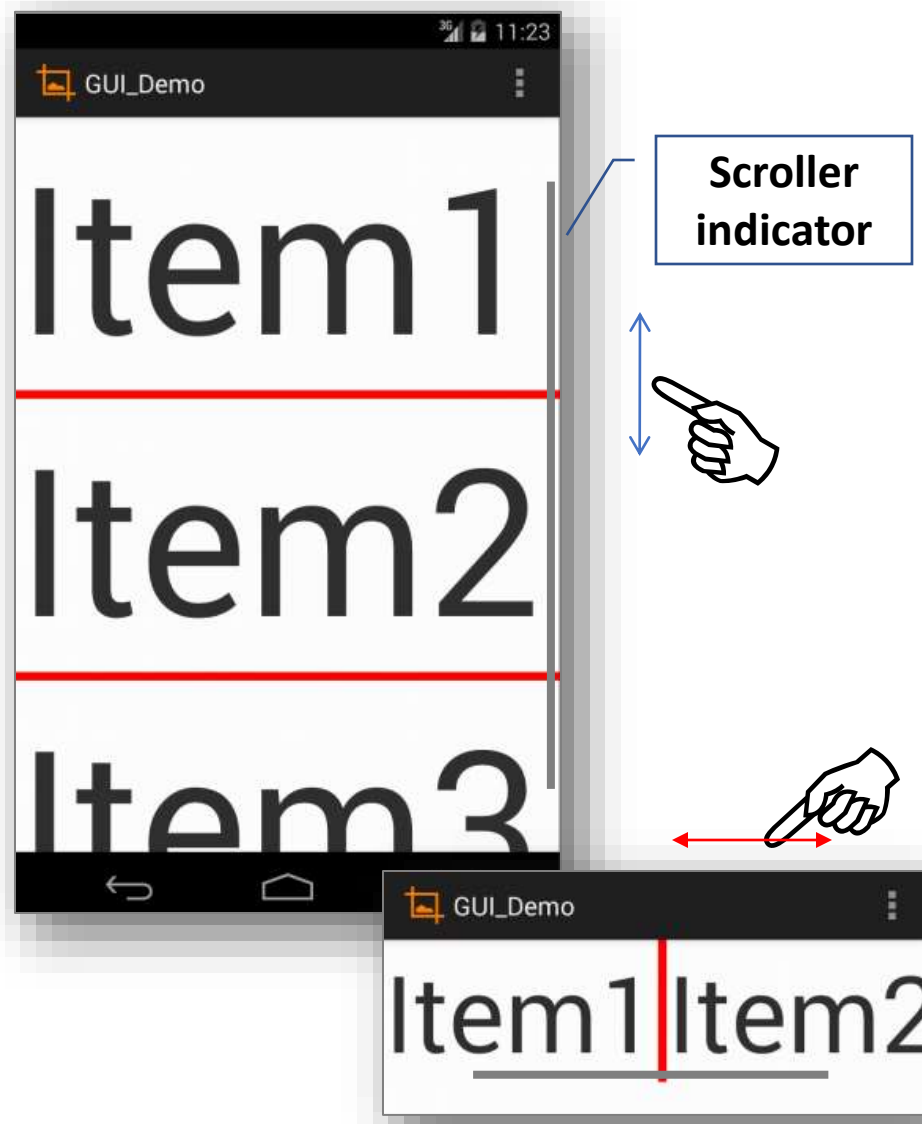
```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/androi
d"
    android:id="@+id/myRelativeLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#ff000099" >

    <TextView
        android:id="@+id/lblUserName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:background="#ffff0066"
        android:text="User Name"
        android:textColor="#ff000000"
        android:textStyle="bold" >
    </TextView>
```




```
    <EditText
        android:id="@+id/txtUserName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/lblUserName"
        android:padding="20dp" >
    </EditText>
    <Button
        android:id="@+id/btnGo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
         android:layout_alignRight="@+id/txtUserName"
        android:layout_below="@+id/txtUserName"
        android:text="Go"
        android:textStyle="bold" >
    </Button>
    <Button
        android:id="@+id/btnCancel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/txtUserName"
        android:layout_toLeftOf="@+id/btnGo"
        android:text="Cancel"
        android:textStyle="bold" >
    </Button>
</RelativeLayout>
```

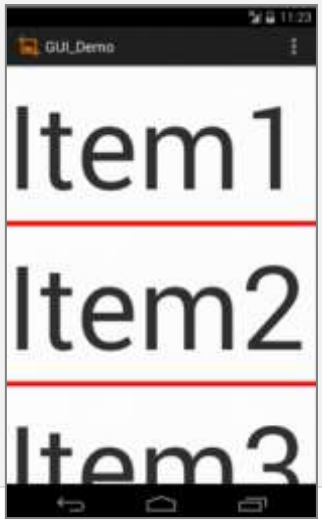
ScrollView Layout (Vertical & Horizontal)

- The **ScrollView** control is useful in situations in which we have *more data to show* than what a single screen could display.
- **ScrollViews** provide a vertical sliding (up/down) access to the data.
- The **HorizontalScrollView** provides a similar left/right sliding mechanism)
- Only a portion of the user's data can be seen at one time, however the rest is available for viewing.



Example: Vertical ScrollView Layout

```
<ScrollView xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:id="@+id/myVerticalScrollView1"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <LinearLayout
       android:id="@+id/myLinearLayoutVertical"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:orientation="vertical" >

       <TextView
           android:id="@+id/textView1"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Item1"
           android:textSize="150sp" />

        <View
            android:layout_width="match_parent"
            android:layout_height="6dp"
            android:background="#ffff0000" />


     <TextView
           android:id="@+id/textView2"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Item2"
           android:textSize="150sp" />

        <View
           android:layout_width="match_parent"
           android:layout_height="6dp"
           android:background="#ffff0000" />

        <TextView
           android:id="@+id/textView3"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Item3"
           android:textSize="150sp" />

    </LinearLayout>

</ScrollView>
```



Example: HorizontalScrollView Layout

```
<HorizontalScrollView

xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/myHorizontalScrollView1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" >

    <LinearLayout
       android:id="@+id/myLinearLayoutVertical"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:orientation="horizontal" >

       <TextView
           android:id="@+id/textView1"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Item1"
           android:textSize="75sp" />
        <View
           android:layout_width="6dp"
           android:layout_height="match_parent"
           android:background="#ffff0000" />
       <TextView
           android:id="@+id/textView2"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Item2"
           android:textSize="75sp" />

       <View
          android:layout_width="6dp"
          android:layout_height="match_parent"
          android:background="#ffff0000" />

       <TextView
           android:id="@+id/textView3"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:text="Item3"
           android:textSize="75sp" />
    </LinearLayout>

</HorizontalScrollView>
```

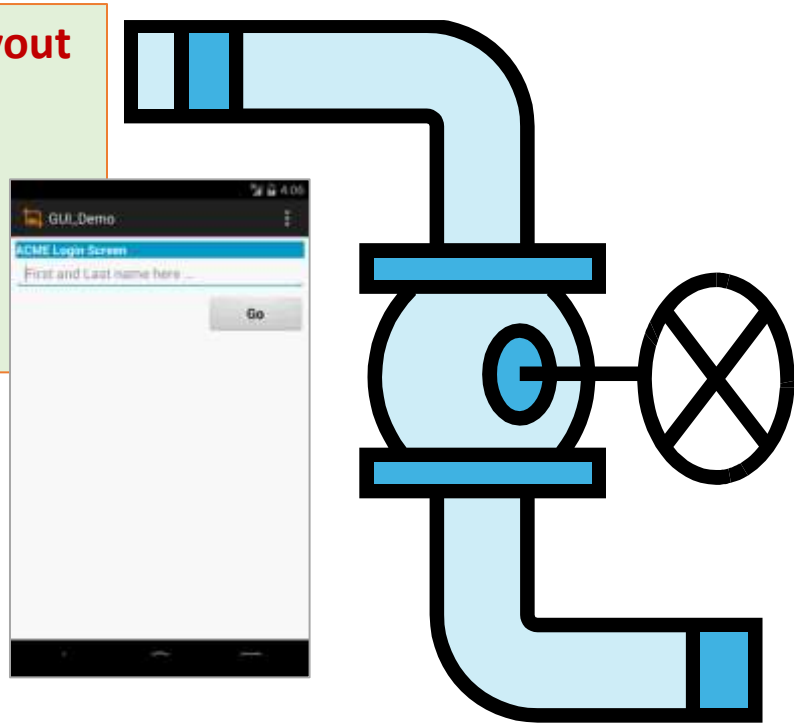
Connecting layouts to Kotlin code

PLUMBING.

You must 'connect' functional XML elements –such as buttons, text boxes, check boxes - with their equivalent Kotlin objects. This is typically done in the **onCreate(…)** method of your main activity. After all the connections are made and programmed, your app should be ready to interact with the user.

XML Layout

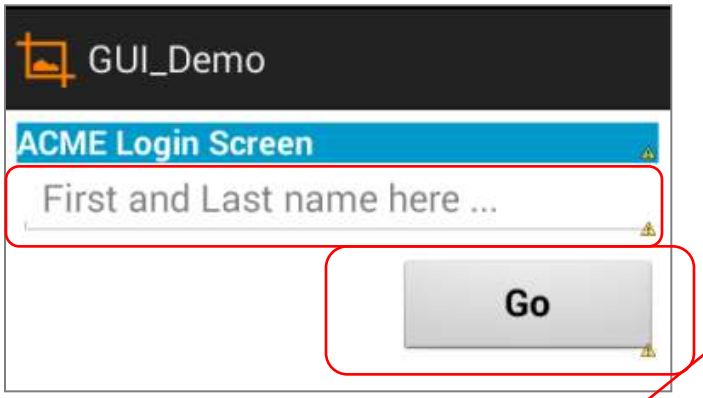
```
<xml….
. . .
. . .
</xml>
```



Kotlin code

```
class MainActivity
{
  …
  …
}
```

Connecting Layouts to Kotlin code




```
<!- XML LAYOUT -->
<LinearLayout
    android:id="@+id/myLinearLayout"
    ... >

    <TextView
        android:text="ACME Login Screen"
        ...  />

    <EditText
        android:id="@+id/edtUserName"
        ... />

    <Button
        android:id="@+id/btnGo"
        ...  />
</LinearLayout>
```

Kotlin code

```
class MainActivity : AppCompatActivity() {

    lateinit var edtUserName: EditText
    lateinit var btnGo: Button

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

        edtUserName = findViewById(R.id.edtUserName)
        btnGo = findViewById(R.id.btnGo)
    }
}
```

What is the meaning of an Android Context?

On Android, a **Context** defines a logical **workspace** on which an app can load and access resources.

- When a widget is created, it is attached to a particular Context. By means of its affiliation to that environment, it then could access other members of the hierarchy on which it has been collocated.

- For a simple '*one activity app*' - say MainActivity - the property **applicationContext** and the reference **MainActivity.this** return the same result.

- An application could have **several activities**. Therefore, for a *multi-activity* app we have one app context, and a context for each of its activities, each good for accessing what is available in *that context.*

Connecting Layouts to Kotlin code

Assume the UI in *res/layout/activity_main.xml* has been created. This layout could be called by an application using the statement

```
setContentView(R.layout.activity_main)
```

Individual XML defined widgets, such as *btnGo* is later associated to the application using the statement `findViewById(...)` as in

```
val btnGo: Button = findViewById(R.id.btnGo)
```

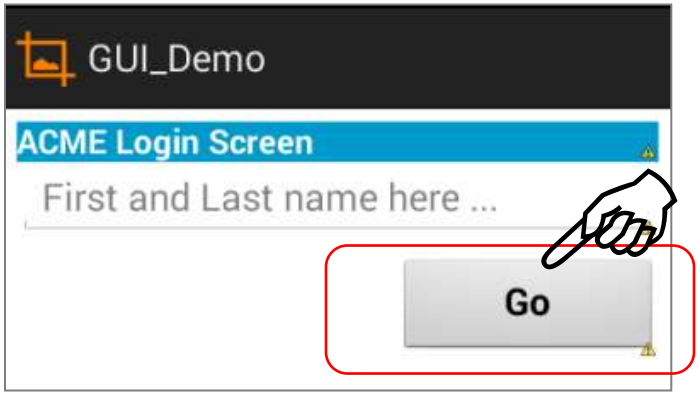
Where **R** is a class automatically generated to keep track of resources available to the application. In particular **R.id...** is the collection of widgets defined in the XML layout (Use Eclipse's Package Explorer, look at your **/gen/package/R.java** contents).

> **A Suggestion**: The widget's identifiers used in the XML layout and Java code could be the same. It is convenient to add a prefix to each identifier indicating its nature. Some options are *txt, btn, edt, rad, chk*, etc. Try to be consistent.

Connecting Layouts to Kotlin code

Attaching Listeners to Widgets

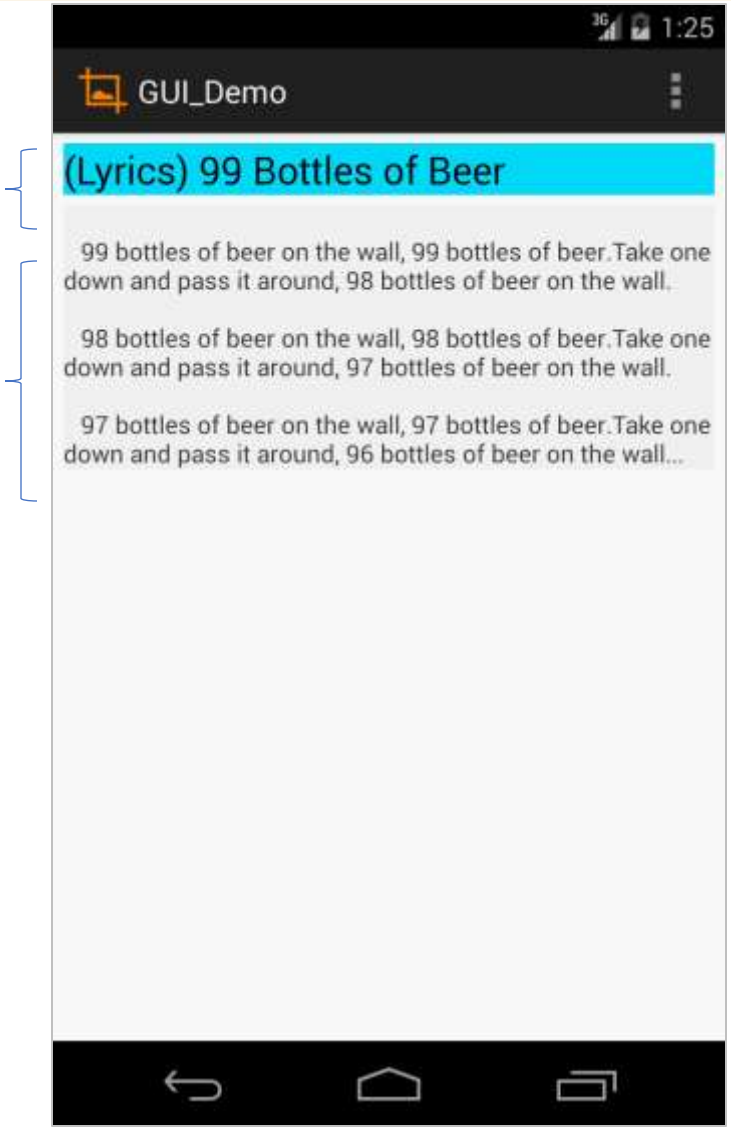
Consider the screen on the right. To make its '**Go'** button widget be responsive to the user's pushing of that button, we may add a listener for the **click event.**




```
val btnGo: Button = findViewById(R.id.btnGo)
btnGo.setOnClickListener {
    // put some more logic here
}
```

Note: Other common 'listeners' watch for events such as: textChanged, tap, long-press, select, focus, etc.

Basic Widgets: TextViews

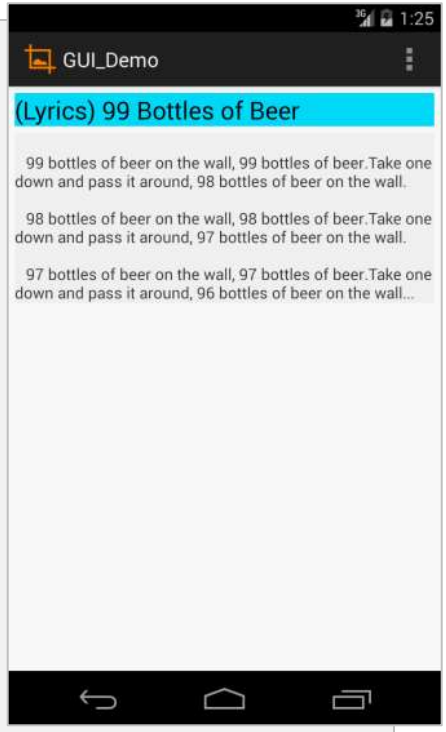


- In Android a **label or text-box** is called a **TextView**.
- A TextView is typically used for showing a caption or a text message.
- TextViews are *not* editable, therefore they take no input.
- The text to be shown may include the **\n** formatting character (newLine)
- You may also use HTML formatting by setting the text to:

```
Html.fromHtml("your html string")
```

Basic Widgets: Example - TextViews

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="6dp" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@color/holo_blue_bright"
        android:text="(Lyrics) 99 Bottles of Beer"
        android:textAppearance="?android:attr/textAppearanceLarge" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="6dp"
        android:background="@color/gray_light"
        android:text="\n\t99 bottles of beer on the wall, 99 bottles of beer.Take one down and
pass it around, 98 bottles of beer on the wall.\n\n\t98 bottles of beer on the wall, 98 bottles
of beer.Take one down and pass it around, 97 bottles of beer on the wall. \n\n\t97 bottles of
beer on the wall, 97 bottles of beer.Take one down and pass it around, 96 bottles of beer on
the wall... "
        android:textSize="14sp" />
</LinearLayout>
```



Basic Widgets: TextViews

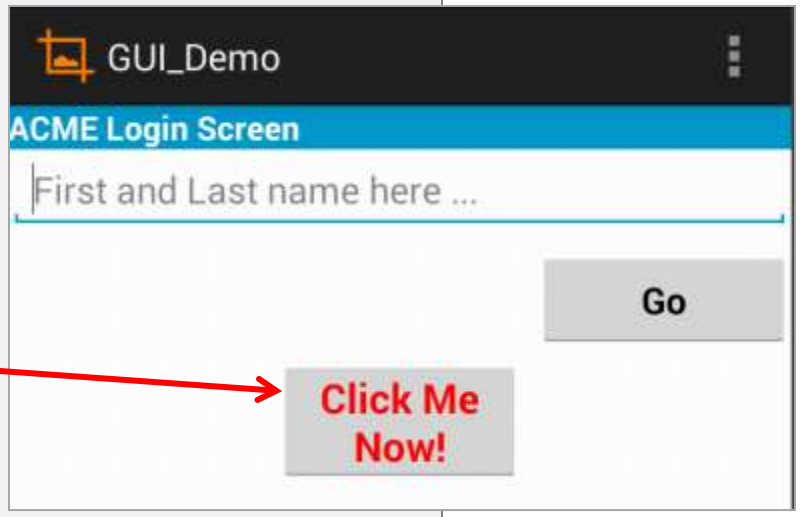
* Set up color (text, background)
  + In XML: RGB, ARGB, from resource
  + In Kotlin code: from resource, from Color class
* Get content from strings.xml resource file
  + In XML: @string
  + In Kotlin code: R.string
* Set custom font from file in the assets folder

```
val textView: TextView = findViewById(R.id.textView)
textView.typeface = Typeface.createFromAsset(assets, "fontname.ttf")
```

Basic Widgets: Buttons

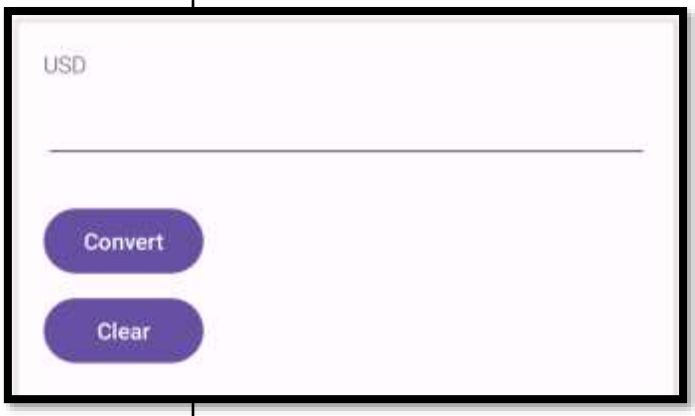
- A **Button** widget allows the simulation of a GUI clicking action.
- **Button** is a subclass of **TextView**. Therefore formatting a button's face is similar to the setting of a **TextView**.
- You may alter the default behavior of a button by providing a custom *drawable.xml* specification to be applied as background. In those specs you indicate the shape, color, border, corners, gradient, and behavior based on states (pressed, focused). More on this issue in the appendix.

```
<Button
        android:id="@+id/btnClickMeNow"
        android:layout_width="120dp"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginTop="5dp“
        android:gravity="center"
        android:padding="5dp"
        android:text="Click Me Now!"
        android:textColor="#ffff0000"
        android:textSize="20sp"
        android:textStyle="bold" />
```



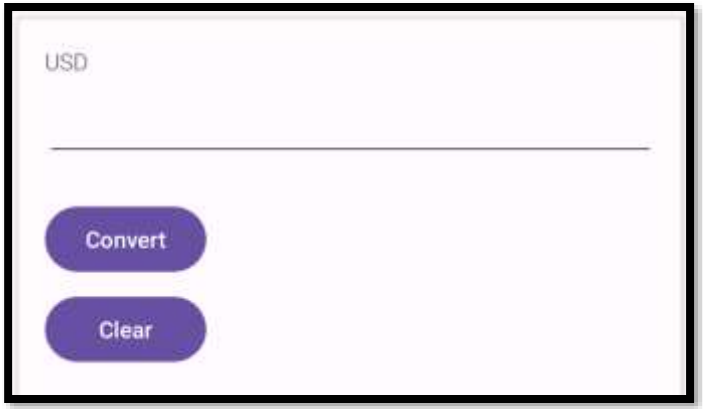
Example: Connecting buttons

```
class MainActivity : AppCompatActivity() {

    lateinit var edtInput: EditText

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

        edtInput = findViewById(R.id.edtInput)

        val btnConvert: Button = findViewById(R.id.btnConvert)
        btnConvert.setOnClickListener {
            // put some more logic here
        }

        val btnClear: Button = findViewById(R.id.btnClear)
        btnClear.setOnClickListener {
            // put some more logic here
        }
    }
}
```



Example: Connecting buttons

```
class MainActivity : AppCompatActivity(), OnClickListener {

    lateinit var edtInput: EditText

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

        edtInput = findViewById(R.id.edtInput)

        val btnConvert: Button = findViewById(R.id.btnConvert)
        btnConvert.setOnClickListener(this)

        val btnClear: Button = findViewById(R.id.btnClear)
        btnClear.setOnClickListener(this)
    }

    override fun onClick(p0: View?) {
        if (p0?.id == R.id.btnConvert) {
            // put some more logic here
        } else if (p0?.id == R.id.btnClear) {
            // put some more logic here
        }
    }
}
```



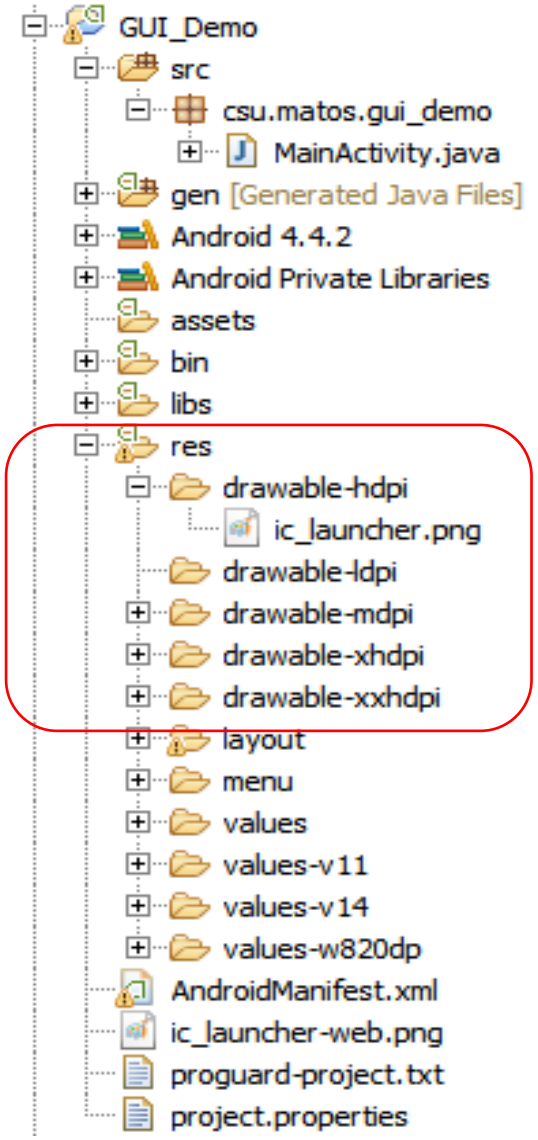
Basic Widgets: ImageView & ImageButton

- **ImageView** and **ImageButton** allow the embedding of images in your applications (gif, jpg, png, etc).
- Analogue to *TextView* and *Button* controls (respectively).
- Each widget takes an

  `android:src` or

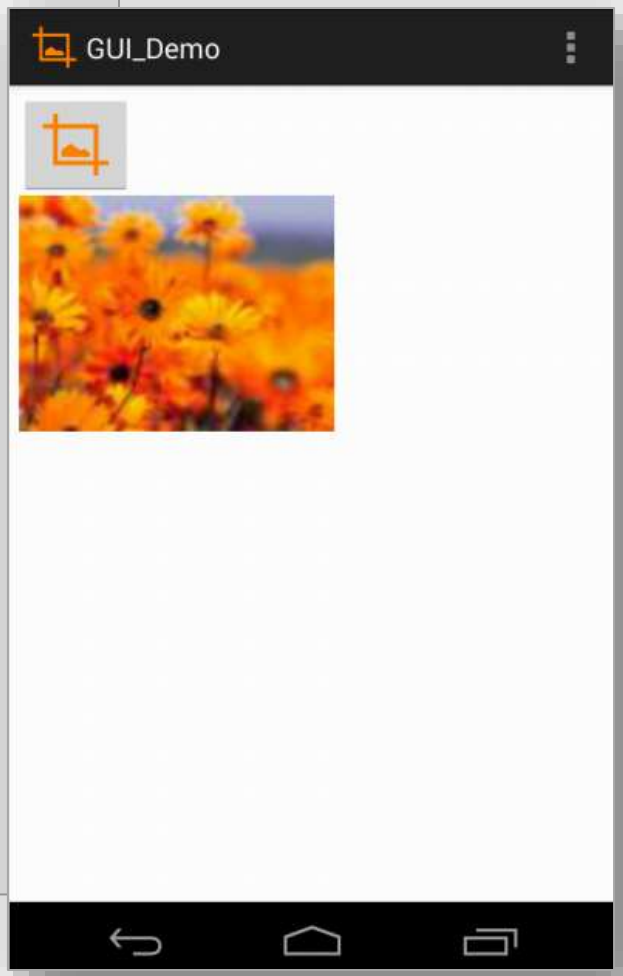
  `android:background`

  attribute (in an XML layout) to specify what picture to use.
- Pictures are stored in the **res/drawable** folder (optionally a *medium, high, x-high, xx-high, and xxx-high* respectively definition version of the same image could be stored for later usage with different types of screens). Details available at: http://developer.android.com/design/style/iconography.html



Basic Widgets: ImageView & ImageButton

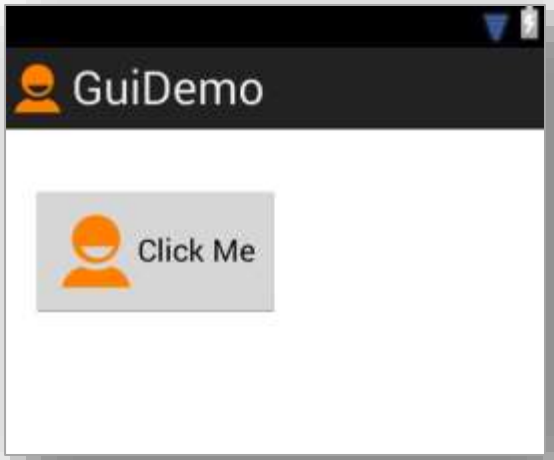
```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="6dp"
    android:orientation="vertical" >

    <ImageButton
        android:id="@+id/imgButton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/ic_launcher" >
    </ImageButton>

    <ImageView
        android:id="@+id/imgView1"
        android:layout_width="200dp"
        android:layout_height="150dp"
        android:scaleType="fitXY"
        android:src="@drawable/flowers1" >
    </ImageView>

</LinearLayout>
```



Basic Widgets: Buttons - Combining Images & Text

A common **Button** widget could display text and a simple image as shown below

```
<LinearLayout
    . . .

 <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:drawableLeft="@drawable/ic_launcher"
        android:gravity="left|center_vertical"
        android:padding="15dp"
        android:text="Click me" />

</LinearLayout>
```



Basic Widgets: How icons are used in Android?

Icons are small images used to graphically represent your application and/or parts of it. They may appear in different parts of your app including:

- Home screen
- Launcher window.
- Options menu
- Action Bar
- Status bar
- Multi-tab interface.
- Pop-up dialog boxes
- List view

Detailed information on Android's iconography is available at: http://developer.android.com/design/style/iconography.html

mdpi (761 bytes)
1x = 48 x 48 pixels
BaseLine

hdpi (1.15KB)
1.5x = 72 x 72 px

x-hdpi (1.52KB)
2x = 96 x 96 px

xx-hdpi (2.47KB)
3x = 144 x 144 px

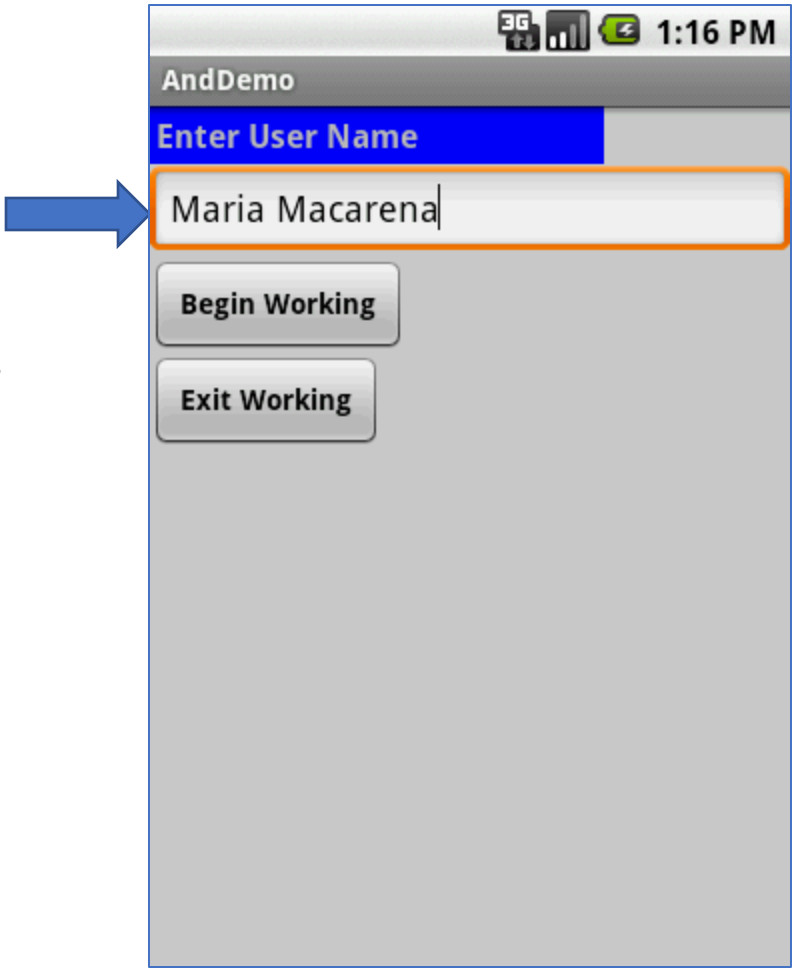
Basic Widgets: EditText Boxes

- The **EditText** widget is an extension of TextView that allows user's input.

- In addition to plain text, this widget can display **editable** text formatted with HTML-styles such as bold, italics, underline, etc). This is done with `Html.fromHtml(html_text)`

- Moving data in and out of an EditText box is usually done in Kotlin through the following methods:

```
txtBox.setText("someValue")

txtBox.text.toString()
```



Basic Widgets: EditText Boxes

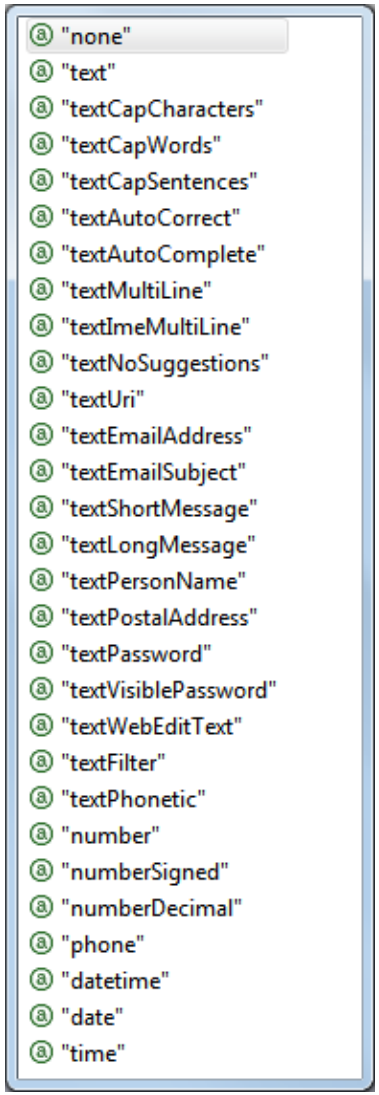
Input Type Formats

An EditText box could be set to accept input strings satisfying a particular pattern such as: numbers (with and without decimals or sign), phones, dates, times, uris, etc.

Setting the EditText box to accept a particular choice of data-type, is done through the XML clause

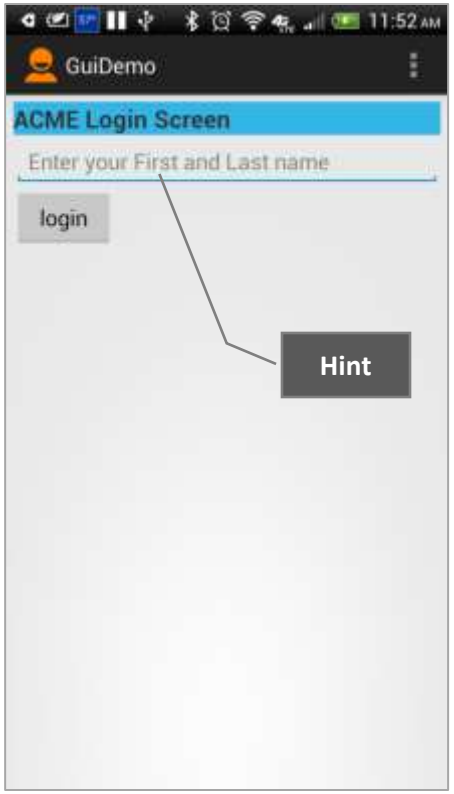
```
android:inputType="choices"
```

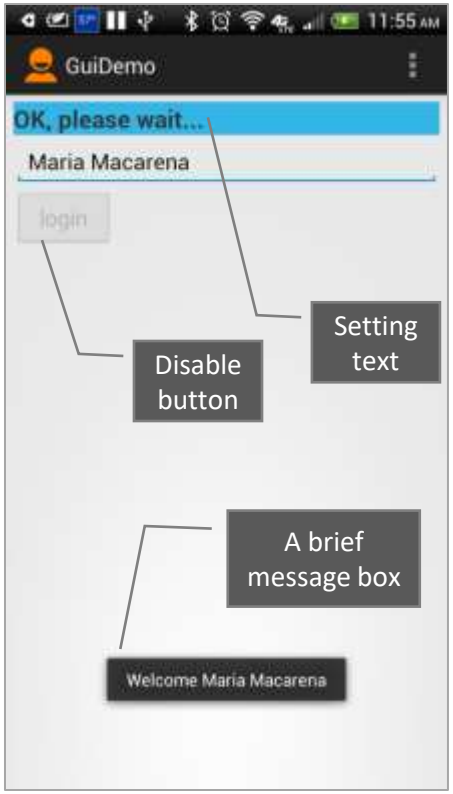
where **choices** include any of the single values shown in the figure. You may combine types, for instance:
`textCapWords|textAutoCorrect`
Accepts text that capitalizes every word, incorrect words are automatically changed (for instance 'teh' is converted into 'the', and so on.



Example: Login-Screen

In this example we will create a simple login screen holding a label (**TexView**), a textBox (**EditText**), and a **Button**. When the EditText box gains focus, the system provides a **virtual keyboard** customized to the input-type given to the entry box (capitals & spelling). Clicking the button displays a Toast-message that echoes the supplied user-name.





Example: Login-Screen

LAYOUT 1 of 2

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="6dp" >

    <TextView
        android:id="@+id/txtLogin"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/holo_blue_light"
        android:text="@string/ACME_Login_Screen"
        android:textSize="20sp"
        android:textStyle="bold" />

    <EditText
        android:id="@+id/edtUserName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="2dp"
        android:hint="@string/Enter_your_First_and_Last_name"
        android:inputType="textCapWords|textAutoCorrect"
        android:textSize="18sp" >
        <requestFocus />
    </EditText>
```

Example: Login-Screen

LAYOUT 2 of 2

```
    <Button
        android:id="@+id/btnLogin"
        android:layout_width="82dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="2dp"
        android:text="@string/login" />
</LinearLayout>
```

res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!-- this is the res/values/strings.xml file -->
<resources>

    <string name="app_name">GuiDemo</string>
    <string name="action_settings">Settings</string>
    <string name="login">login</string>
    <string name="ACME_Login_Screen">ACME Login Screen</string>
    <string name="Enter_your_First_and_Last_name">Enter your First and Last name</string>

</resources>
```

Example: Login-Screen - MainActivity code

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

        val context = applicationContext
        val duration = Toast.LENGTH_SHORT

        val textLogin: TextView = findViewById(R.id.text_login)
        val editUsername: EditText = findViewById(R.id.edit_username)

        val buttonLogin: Button = findViewById(R.id.button_login)
        buttonLogin.setOnClickListener {
            val username = editUsername.text.toString()

            Log.v("TAG", "duration: $duration")
            Log.v("TAG", "context: $context")
            Log.v("TAG", "username: $username")

            if (username.equals("Maria Macarena")) {
                textLogin.text = "OK, please wait"
                Toast.makeText(this, "Welcome $username", duration).show()
                buttonLogin.isEnabled = false
            } else {
                Toast.makeText(this, "$username is not a valid user", duration).show()
            }
        }
    }
}
```

Log.v used for debugging, show in Logcat window

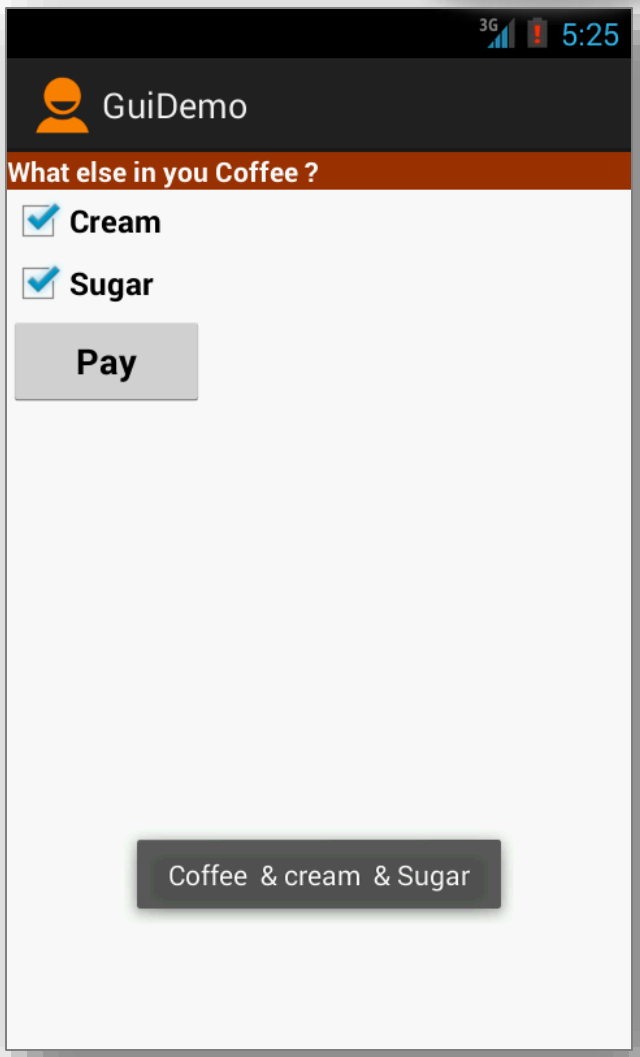
Basic Widgets: CheckBoxes

A checkbox is a special **two-states** button which can be either *checked* or *unchecked*.

A screen may include any number of **mutually inclusive** (independent) CheckBoxes. At any time, more than one CheckBox in the GUI could be checked.

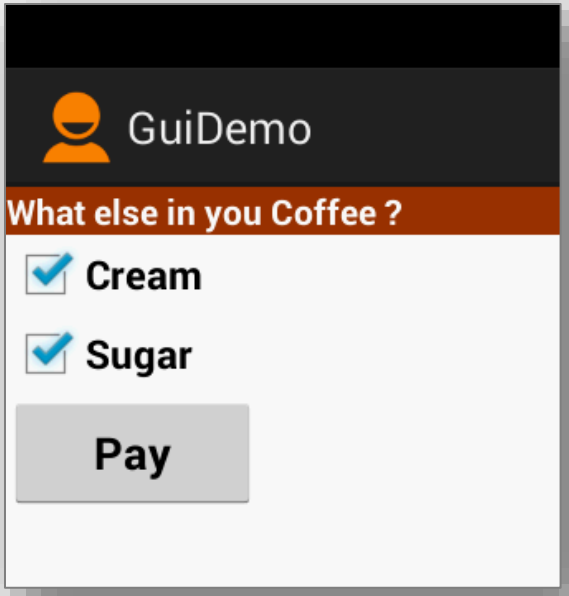
In our "CaféApp" example, the screen on the right displays two CheckBox controls, they are used for selecting 'Cream' and 'Sugar' options. In this image both boxes are 'checked'.

When the user pushes the 'Pay' button a Toast-message is issue echoing the current combination of choices held by the checkboxes.



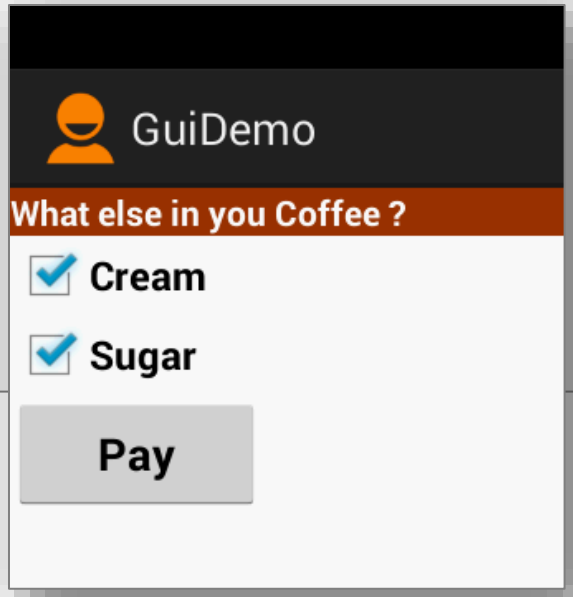
Example: CheckBoxes – CaféApp [Layout 1 of 2]

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="6dp"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/labelCoffee"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#ff993300"
        android:text="@string/coffee_addons"
        android:textColor="@android:color/white"
        android:textStyle="bold" />

    <CheckBox
        android:id="@+id/chkCream"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/cream"
        android:textStyle="bold" />
```



Example: CheckBoxes – CaféApp [Layout 2 of 2]

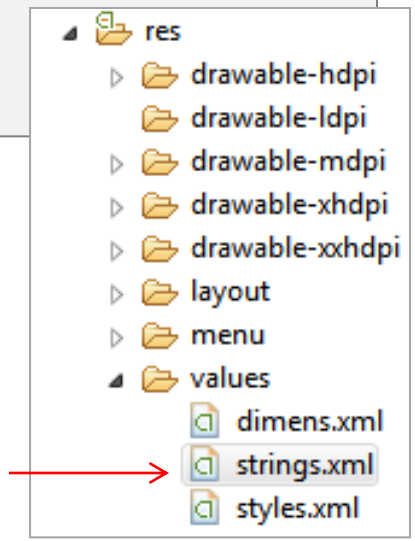
```
    <CheckBox
        android:id="@+id/chkSugar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/sugar"
        android:textStyle="bold" />

    <Button
        android:id="@+id/btnPay"
        android:layout_width="153dp"
        android:layout_height="wrap_content"
        android:text="@string/pay"
        android:textStyle="bold" />

</LinearLayout>
```



Example: CheckBoxes – CaféApp [@string/…]

Resources: res/values/strings

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <string name="app_name">GuiDemo</string>
    <string name="action_settings">Settings</string>

    <string name="click_me">Click Me</string>
    <string name="sugar">Sugar</string>
    <string name="cream">Cream</string>
    <string name="coffee_addons">What else in your coffee?</string>
    <string name="pay">Pay</string>
</resources>
```



Example: CheckBoxes – CaféApp [Code]

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

        val chkCream: CheckBox = findViewById(R.id.chkCream)
        val chkSugar: CheckBox = findViewById(R.id.chkSugar)

        val buttonLogin: Button = findViewById(R.id.btnPay)
        buttonLogin.setOnClickListener {
            var msg = "Coffee "
            if (chkCream.isChecked)
                msg += "& cream "
            if (chkSugar.isChecked)
                msg += "& sugar "
            Toast.makeText(applicationContext, msg, Toast.LENGTH_SHORT).show()
        }
    }
}
```

Basic Widgets: Radio Buttons

- A **radio button** (like a CheckBox) is a two-states button that can be either *checked* or *unchecked*.

- Logically related radio buttons are normally put together in a **RadioGroup** container. The container forces the enclosed radio buttons to behave as **mutually exclusive selectors.** That is, the checking of one radio button *unchecks* all the others.

- Properties for *font face, style, color,* etc. are managed in a way similar to setting a TextView.

- You may call the method ***isChecked()*** to see if a specific RadioButton is selected, or change its state by calling **toggle()**.

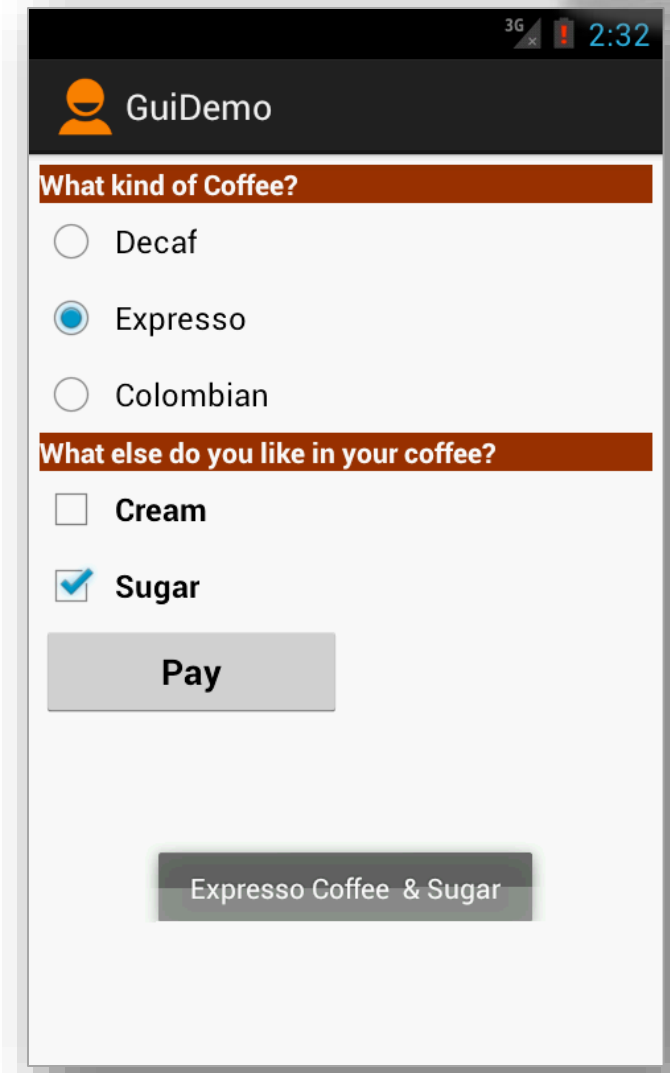
Example: CheckBoxes – CaféApp [Layout]

Example

*We extend the previous CaféApp example by adding a **RadioGroup** control that allows the user to pick one type of coffee from three available options.*

RadioGroup

Summary of choices



Example: CheckBoxes – CaféApp [Layout]

Only new XML and Kotlin code is shown

```
<TextView
      android:id="@+id/textView1"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:background="#ff993300"
      android:text="@string/kind_of_coffee"
      android:textColor="#ffffff"
      android:textStyle="bold" />
```


```
<RadioGroup
      android:id="@+id/radioGroupCoffeeType"
      android:layout_width="match_parent"
      android:layout_height="wrap_content" >

      <RadioButton
          android:id="@+id/radDecaf"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/decaf" />

      <RadioButton
          android:id="@+id/radExpresso"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:text="@string/expresso" />

      <RadioButton
          android:id="@+id/radColombian"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:checked="true"
          android:text="@string/colombian" />
  </RadioGroup>
```

Example: CheckBoxes – CaféApp [MainActivity]

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

        val chkCream: CheckBox = findViewById(R.id.chkCream)
        val chkSugar: CheckBox = findViewById(R.id.chkSugar)

        val radCoffeeType: RadioGroup = findViewById(R.id.radioGroupCoffeeType)
        val radDecaf: RadioButton = findViewById(R.id.radDecaf)
        val radExpresso: RadioButton = findViewById(R.id.radExpresso)
        val radColombian: RadioButton = findViewById(R.id.radColombian)

        val buttonLogin: Button = findViewById(R.id.btnPay)
        buttonLogin.setOnClickListener {
            var msg = "Coffee "
            if (chkCream.isChecked)
                msg = "$msg & cream "
            if (chkSugar.isChecked)
                msg = "$msg & sugar "

            val radioId = radCoffeeType.checkedRadioButtonId

            if (radColombian.id == radioId) msg = "Colombian $msg"
            if (radExpresso.isChecked) msg = "Expresso $msg"
            if (radDecaf.isChecked) msg = "Decaf $msg"
```

Example: CheckBoxes – CaféApp [MainActivity]

```
            // Alternative
            when (radioId) {
                R.id.radColombian -> msg = "Colombian $msg"
                R.id.radExpresso -> msg = "Expresso $msg"
                R.id.radDecaf -> msg = "Decaf $msg"
            }

            Toast.makeText(applicationContext, msg, Toast.LENGTH_SHORT).show()
        }
    }
}
```

Alternative you may also manage a **RadioGroup** as follows (this is simpler because you don't need to define the individual RadioButtons

Appendix. Using the @string resource

A *good programming practice* in Android is **NOT** to directly enter literal strings as immediate values for attribute inside xml files.
For example, if you are defining a **TextView** to show a company headquarter's location, a clause such as `android:text="Cleveland"` should not be used (observe it produces a **Warning** *[I18N] Hardcoded string "Cleveland", should use @string resource*)

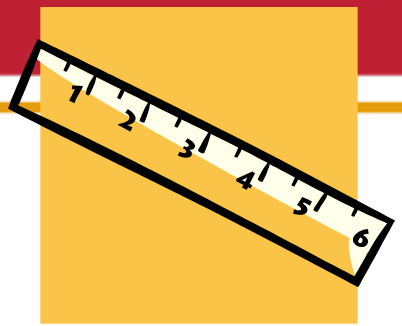
Instead you should apply a two steps procedure in which

1. You write the literal string –say *headquarter* – in **res/values/string.xml.** Enter
   `<string name="headquarter">Cleveland</string>`

2. Whenever the string is needed provide a reference to the string using the notation *@string/headquarter*. For instance in our example you should enter
   `android:text="@string/headquarter"`

WHY?
If the string is used in many places and its actual value changes we just update the resource file entry once. It also provides some support for internationalization -easy to change a resource string from one language to another.

Appendix. Measuring Graphic Elements



Q. What is **dpi** (also know as **dp** and **ppi**) ?

Stands for *dots per inch*. It suggests a measure of screen quality. You can compute it using the following formula:

$$dpi = \sqrt{widthPixels^2 + heightPixels^2} \ / \ diagonalInches$$

G1 (base device 320x480)	155.92 dpi	(3.7 in diagonally)
Nexus (480x800)	252.15 dpi	
HTC One (1080x1920)	468 dpi	(4.7 in)
Samsung S4 (1080x1920)	441 dpi	(5.5 in)

Q. What is the difference between **dp, dip** and **sp** units in Android?

dp *Density-independent Pixels* – is an abstract unit based on the physical density of the screen. These units are relative to a 160 dpi screen, so one dp is one pixel on a 160 dpi screen. Use it for measuring anything but fonts.

sp

Scale-independent Pixels – similar to the relative density dp unit, but used for **font** size preference.

Appendix. Measuring Graphic Elements

How Android deals with screen resolutions?

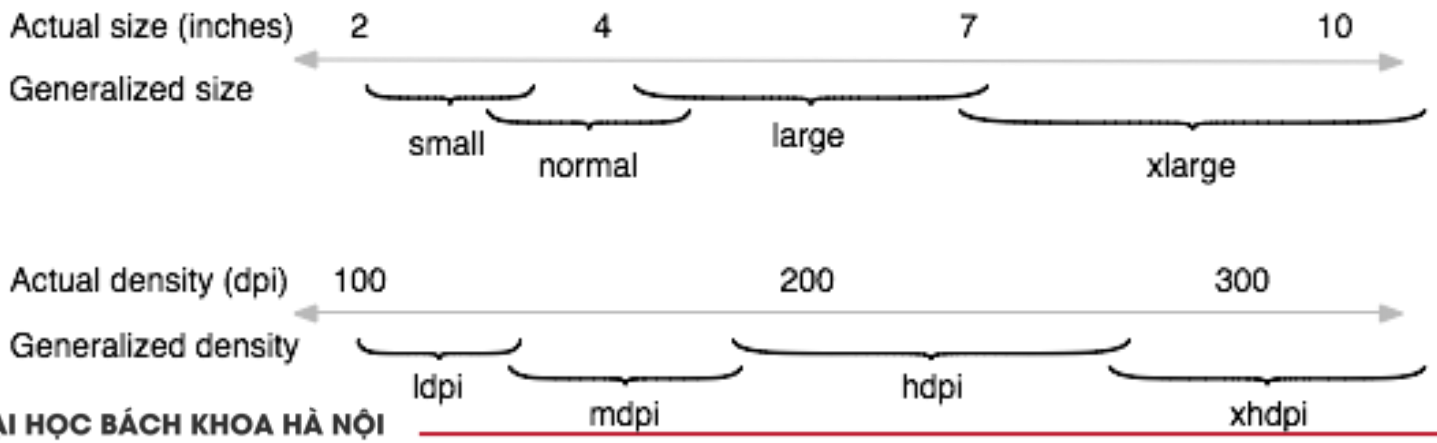
Illustration of how the Android platform maps actual screen densities and sizes to generalized density and size configurations.

A set of four generalized **screen sizes**

xlarge	screens are at least 960dp x 720dp
large	screens are at least 640dp x 480dp
normal	screens are at least 470dp x 320dp
small	screens are at least 426dp x 320dp

A set of six generalized **densities**:

ldpi	~120dpi (low)
mdpi	~160dpi (medium)
hdpi	~240dpi (high)
xhdpi	~320dpi (extra-high)
xxhdpi	~480dpi (extra-extra-high)
Xxxhdpi	~640dpi (extra-extra-extra-high)

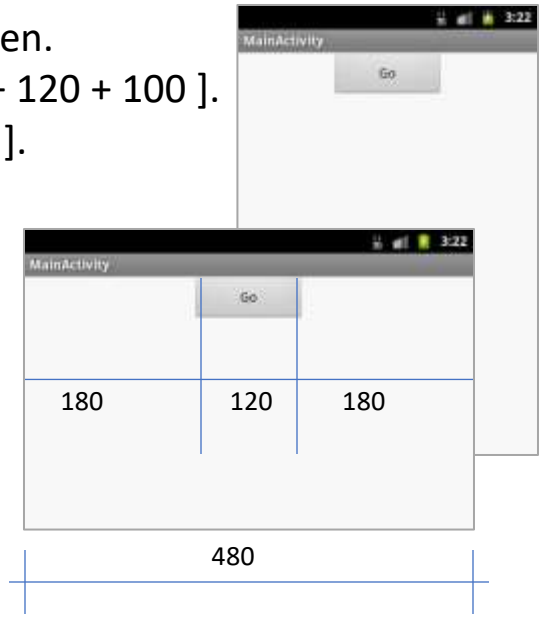


Appendix. Measuring Graphic Elements

Q. Give me an example on how to use dp units.

Assume you design your interface for a G1 phone having 320x480 pixels (Abstracted density is **160** – See your AVD entry, the actual pixeling is defined as: [2*__160]__ x [3*__160]__)

Assume you want a 120dp button to be placed in the middle of the screen.
On portrait mode you could allocate the 320 horizontal pixels as [100 + 120 + 100].
On Landscape mode you could allocate 480 pixels as [180 + 120 + 180].



The XML would be

```
<Button
    android:id="@+id/button1"
    android:layout_height="wrap_content"
    android:layout_width="120dp"
    android:layout_gravity="center"
    android:text="@+id/go_caption" />
```

If the application is deployed on devices having a higher resolution the button is still mapped to the middle of the screen.

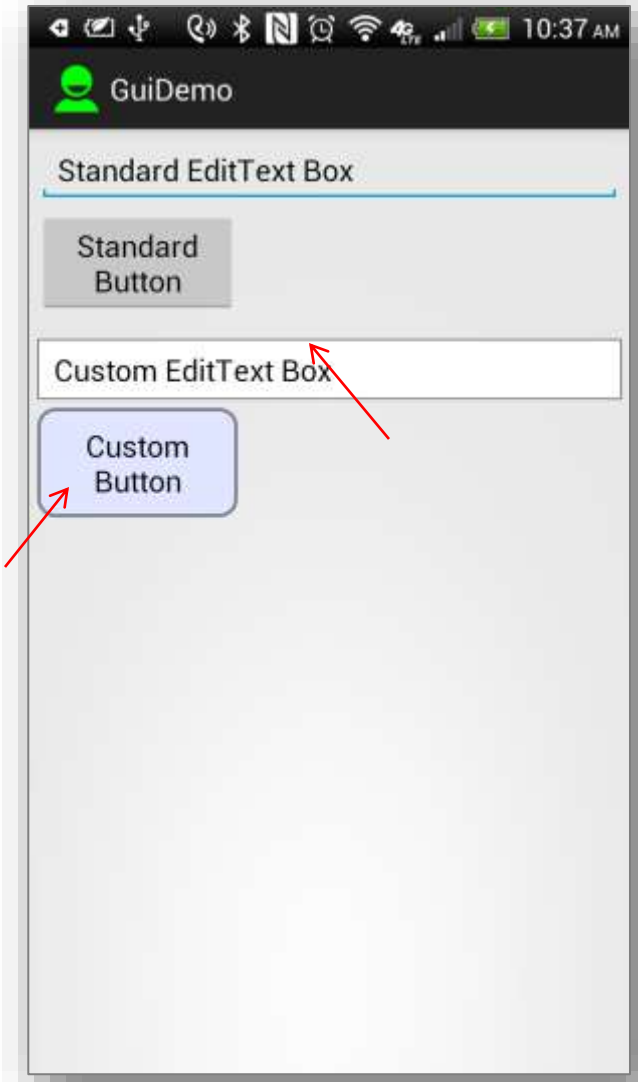
Appendix. Customizing Widgets

1. The appearance of a widget can be adjusted by the user. For example a button widget could be modified by changing its shape, border, color, margins, etc.

2. Basic shapes include: rectangle, oval, line, and ring.

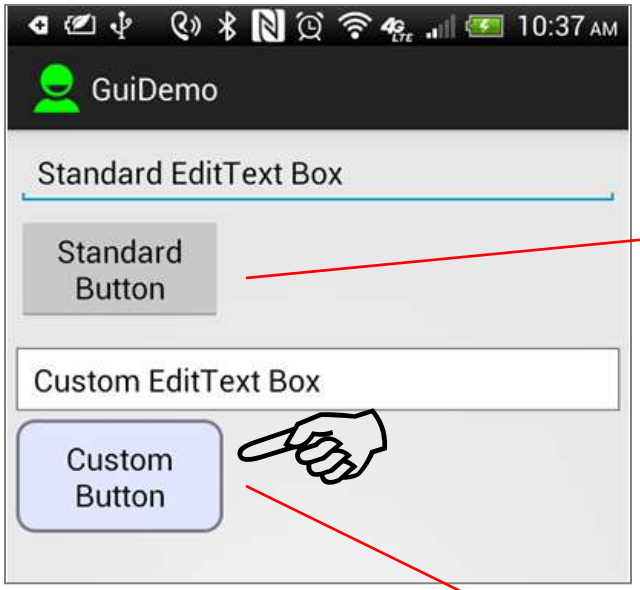
3. In addition to visual changes, the widget's reaction to user interaction could be adjusted for events such as: Focused, Clicked, etc.

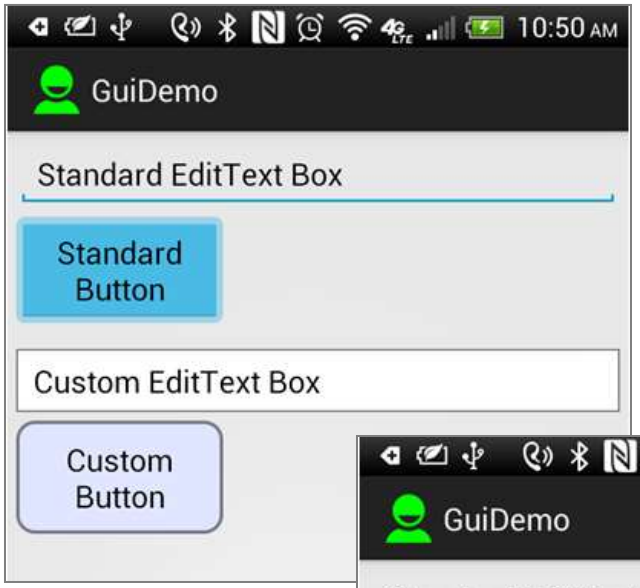
4. The figure shows and EditText and Button widgets as *normally* displayed by a device running SDK4.3 (Ice Cream). The bottom two widgets (a TextView and a Button) are custom made versions of those two controls respectively.



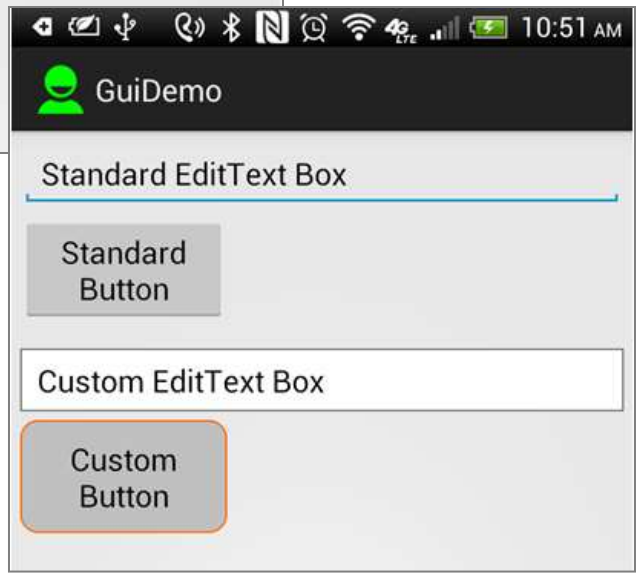
Appendix. Customizing Widgets

The image shows visual feedback provided to the user during the clicking of a standard and a *custom* Button widget.





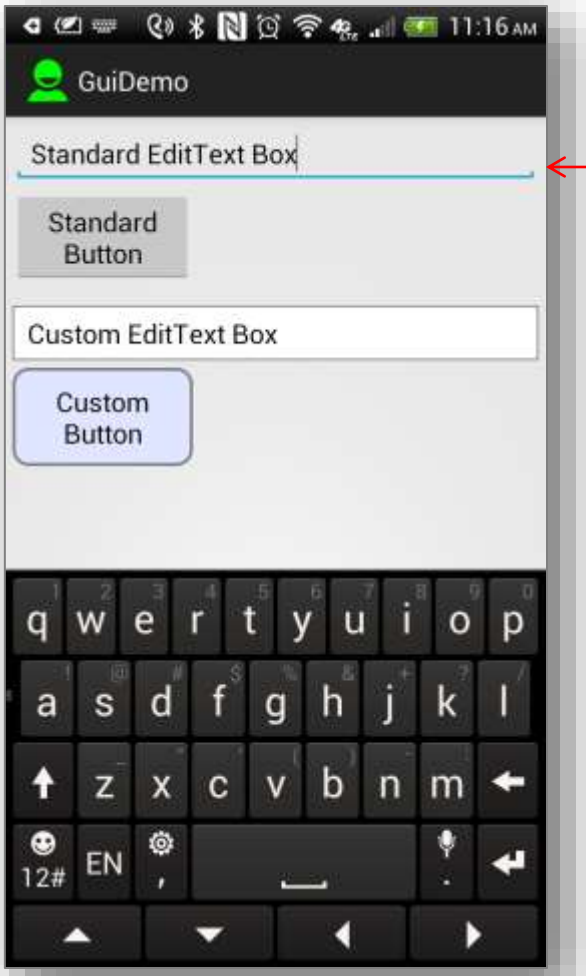
Standard behavior – buttons turns **blue** when it is pressed.



Custom behavior – buttons turns dark grey with an orange border when it is pressed.

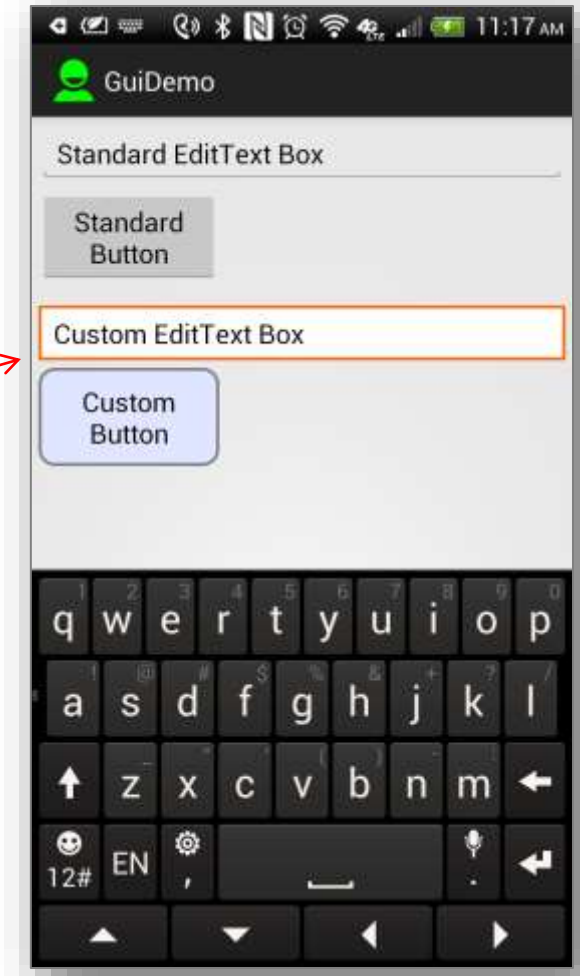
Appendix. Customizing Widgets

Observe the transient response of the standard and custom made EditText boxes when the user touches the widgets provoking the 'Focused' event.



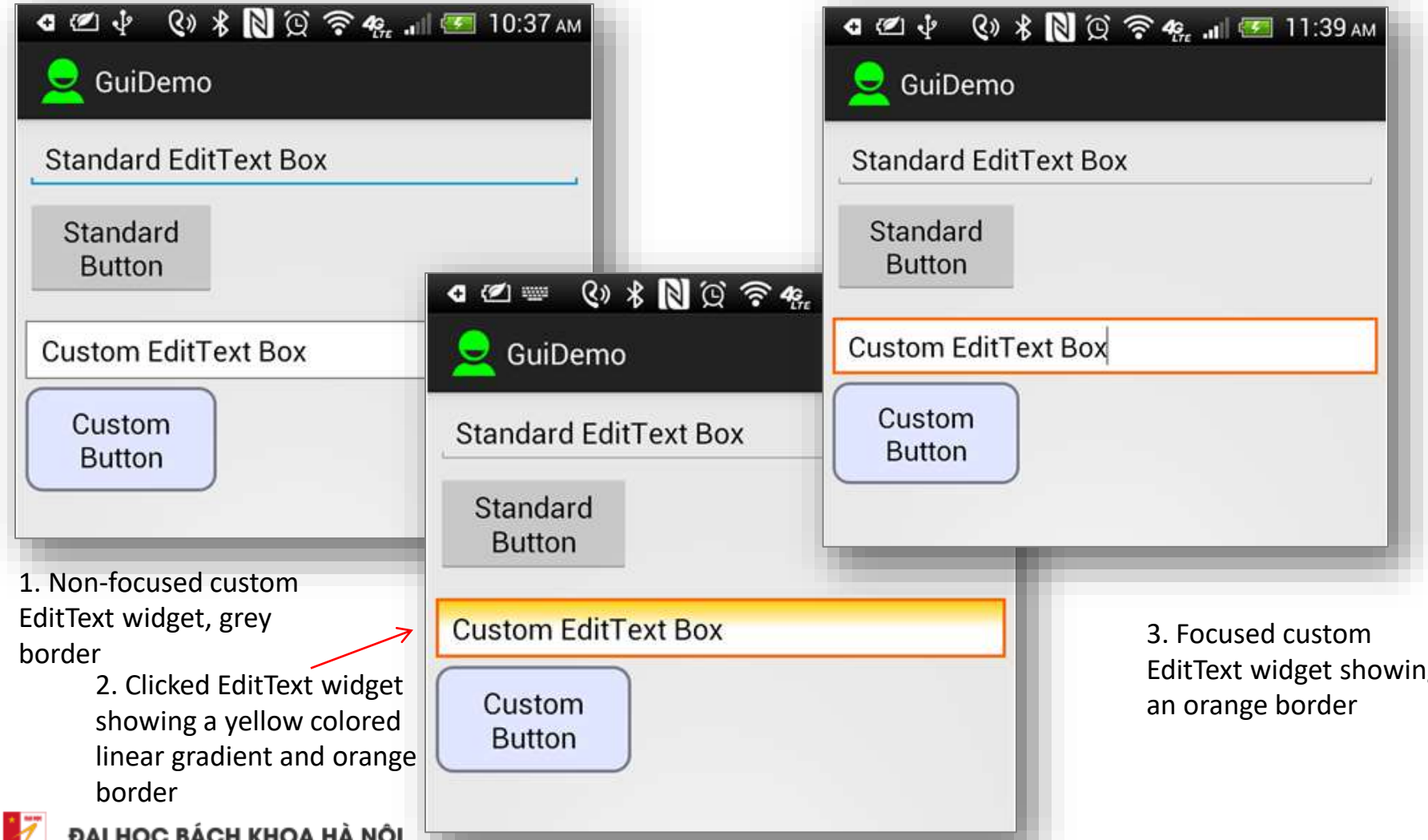
When focused the standard box shows a blue bottom line

A focused custom box shows an orange all-around frame



Appendix. Customizing Widgets

When the user taps on the custom made EditText box a gradient is applied to the box to flash a visual feedback reassuring the user of her selection.



1. Non-focused custom EditText widget, grey border

2. Clicked EditText widget showing a yellow colored linear gradient and orange border

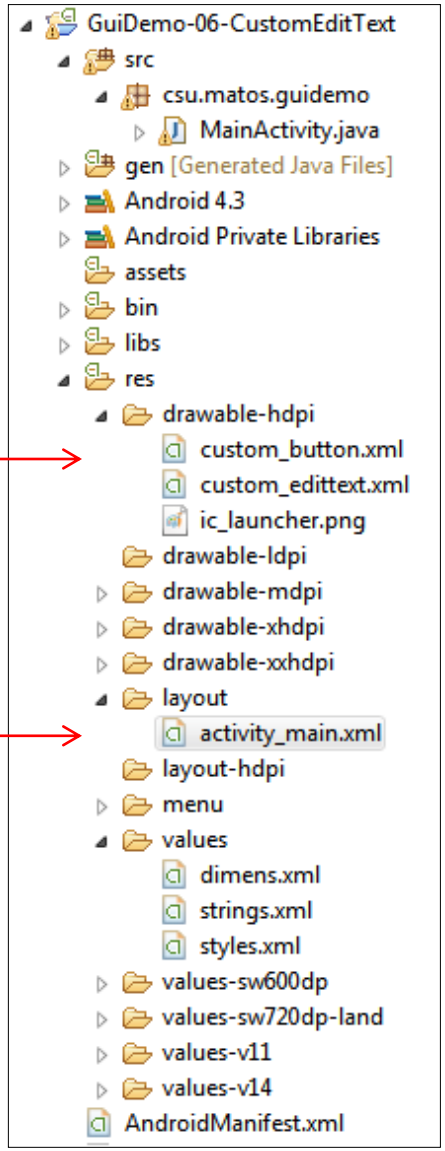
3. Focused custom EditText widget showing an orange border

Appendix. Customizing Widgets

Organizing the application

Definition of the custom templates for Button and EditText widgets

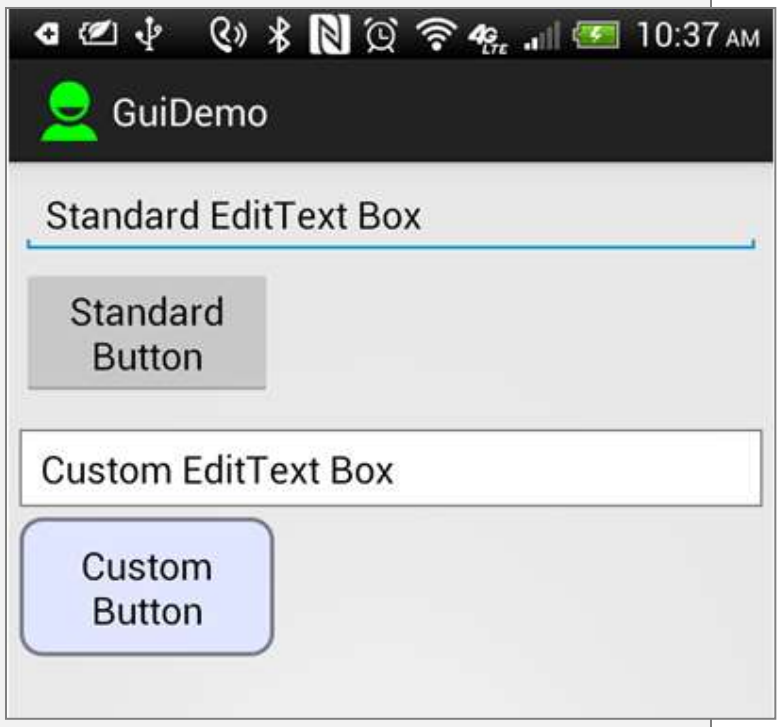
Layout referencing standard and custom made widgets



Appendix. Customizing Widgets

Activity Layout 1 of 2

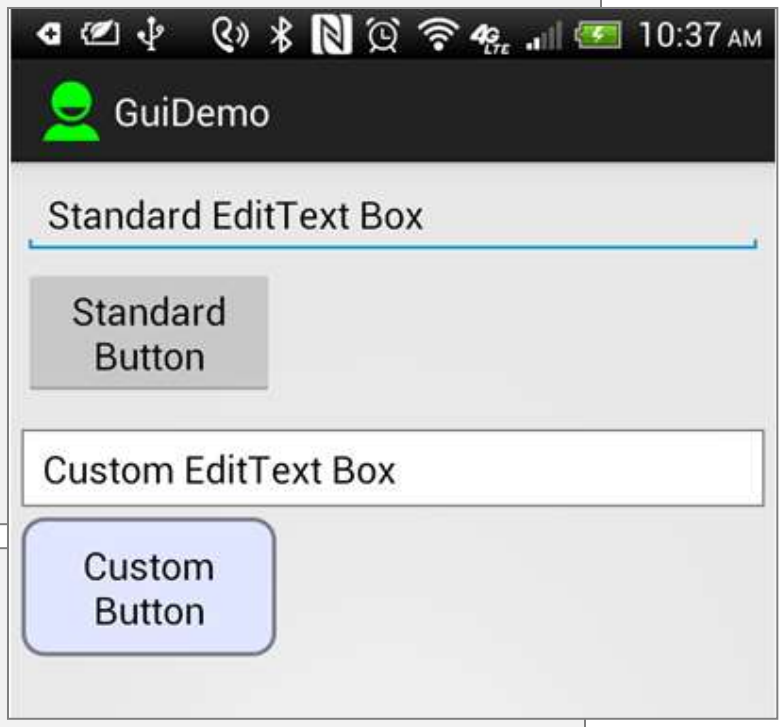
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="5dp" >

    <EditText
        android:id="@+id/editText1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="5dp"
        android:ems="10"
        android:inputType="text"
        android:text="@string/standard_edittext" >

        <requestFocus />
    </EditText>

    <Button
        android:id="@+id/button1"
        android:layout_width="120dp"
        android:layout_height="wrap_content"
        android:layout_marginBottom="15dp"
        android:text="@string/standard_button" />
```



Appendix. Customizing Widgets

Activity Layout (2 of 2) and Resource: res/values/strings

```
    <EditText
        android:id="@+id/editText2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="5dp"
        android:background="@drawable/custom_edittext"
        android:ems="10"
        android:inputType="text"
        android:text="@string/custom_edittext" />

    <Button
        android:id="@+id/button2"
        android:layout_width="120dp"
        android:layout_height="wrap_content"
        android:background="@drawable/custom_button"
        android:text="@string/custom_button" />

</LinearLayout>
```



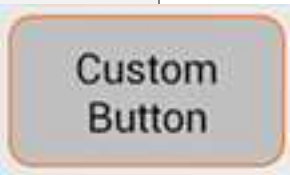

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">GuiDemo</string>
    <string name="action_settings">Settings</string>
    <string name="standard_button">Standard Button</string>
    <string name="standard_edittext">Standard EditText Box</string>
    <string name="custom_button">Custom Button</string>
    <string name="custom_edittext">Custom EditText Box</string>
</resources>
```

Appendix. Customizing Widgets

Resource: res/drawable/custom_button.xml

The custom Button widget has two faces based on the event **state_pressed** (true, false). The Shape attribute specifies its solid color, padding, border (stroke) and corners (rounded corners have radius > 0)

```xml
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
    <item android:state_pressed="true">
        <shape android:shape="rectangle">
            <corners   android:radius="10dp"/>
            <solid     android:color="#ffc0c0c0" />
            <padding   android:left="10dp"
                       android:top="10dp"
                       android:right="10dp"
                       android:bottom="10dp"/>
            <stroke android:width="1dp" android:color="#ffFF6600"/>
        </shape>
    </item>
    <item android:state_pressed="false">
        <shape android:shape="rectangle">
            <corners   android:radius="10dp"/>
            <solid     android:color="#ffE0E6FF"/>
            <padding   android:left="10dp"
                       android:top="10dp"
                       android:right="10dp"
                       android:bottom="10dp"/>
            <stroke android:width="2dp" android:color="#ff777B88"/>
        </shape>
    </item>
</selector>
```



Appendix. Customizing Widgets

Resource: res/drawable/custom_edittext.xml

The rendition of the custom made EditText widget is based on three states: normal, state_focused, state_pressed.

```xml
<?xml version="1.0" encoding="utf-8"?>
<selector  xmlns:android="http://schemas.android.com/apk/res/android">

<item android:state_pressed="true">
    <shape android:shape="rectangle">
          <gradient
               android:angle="90"
               android:centerColor="#FFffffff"
               android:endColor="#FFffcc00"
               android:startColor="#FFffffff"
               android:type="linear" />

        <stroke   android:width="2dp"
                  android:color="#FFff6600" />
        <corners  android:radius="0dp" />
        <padding  android:left="10dp"
                  android:top="6dp"
                  android:right="10dp"
                  android:bottom="6dp" />
    </shape>
</item>
```

Custom EditText Box

Appendix. Customizing Widgets

Resource: res/drawable/custom_edittext.xml

The rendition of the custom made EditText widget is based on three states: normal, state focused, state_pressed.

```
<item android:state_focused="true">
    <shape>
        <solid   android:color="#FFffffff" />
        <stroke  android:width="2dp" android:color="#FFff6600" />
        <corners android:radius="0dp" />
        <padding android:left="10dp"
                 android:top="6dp"
                 android:right="10dp"
                 android:bottom="6dp" />
    </shape>
</item>

<item>
    <!-- state: "normal" not-pressed & not-focused  -->
    <shape>
        <stroke  android:width="1dp" android:color="#ff777777" />
        <solid     android:color="#ffffffff" />
        <corners android:radius="0dp" />
        <padding android:left="10dp"
                 android:top="6dp"
                 android:right="10dp"
                 android:bottom="6dp" />
    </shape>
</item>
</selector>
```

Custom EditText Box

Custom EditText Box