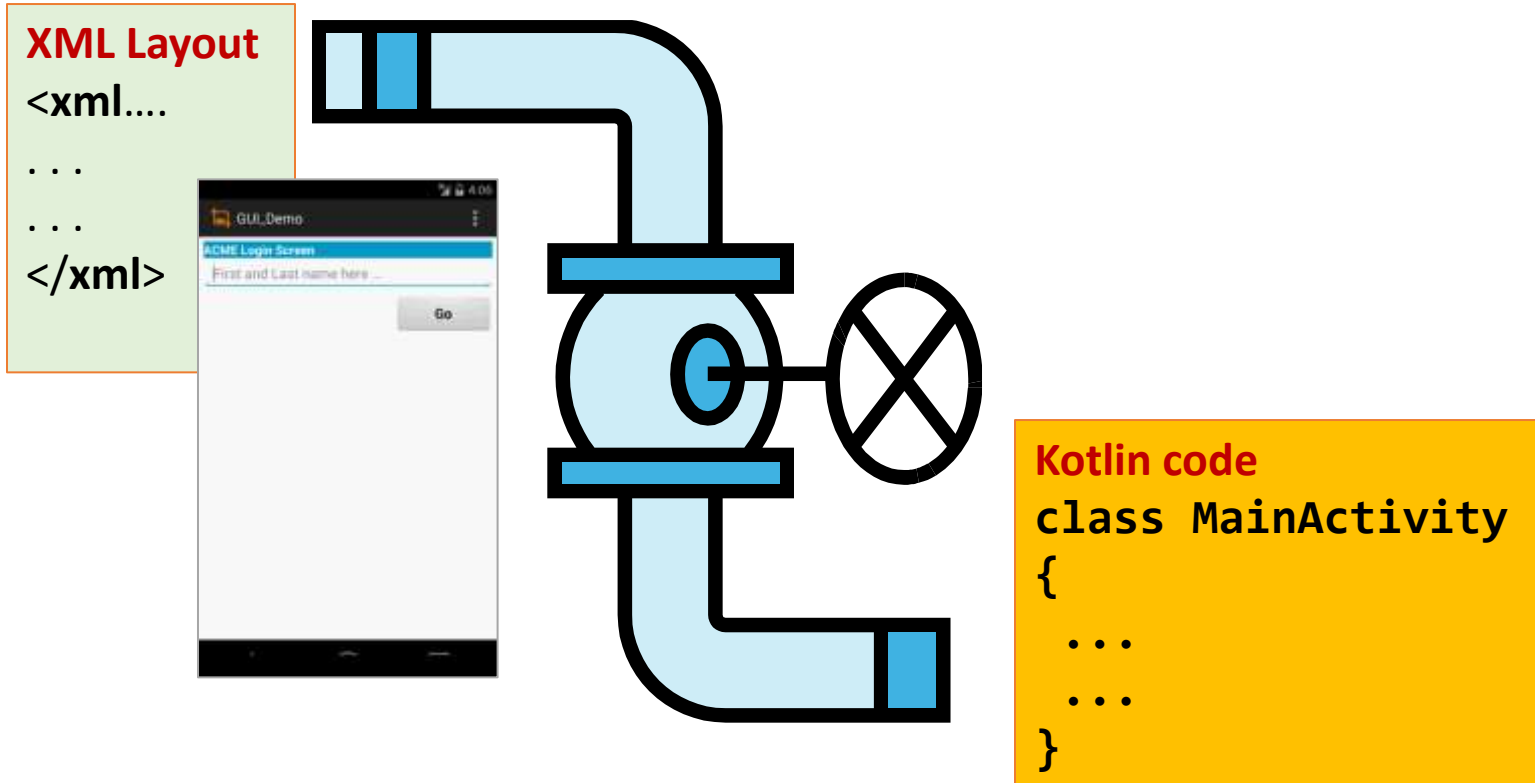


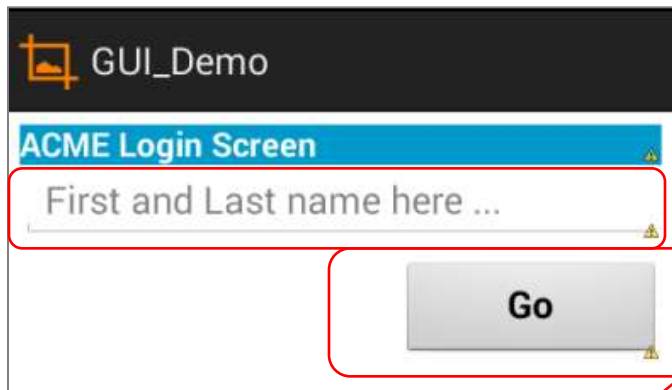
Connecting layouts to Kotlin code

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.



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)
```

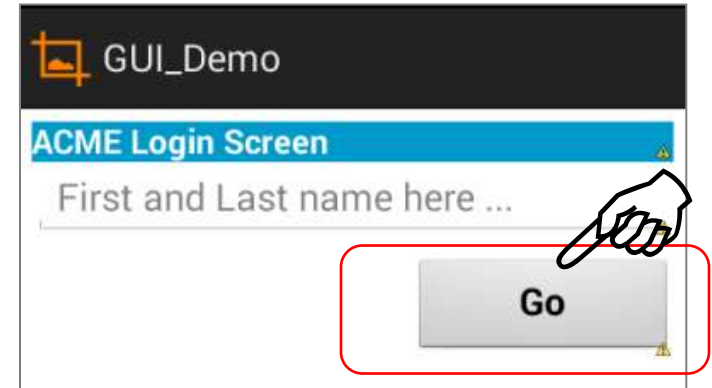
Where **R** is a class automatically generated to keep track of resources available to the application. In particular **R.id...** is the collection of widgets defined in the XML layout (Use Eclipse's Package Explorer, look at your **/gen/package/R.java** contents).

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

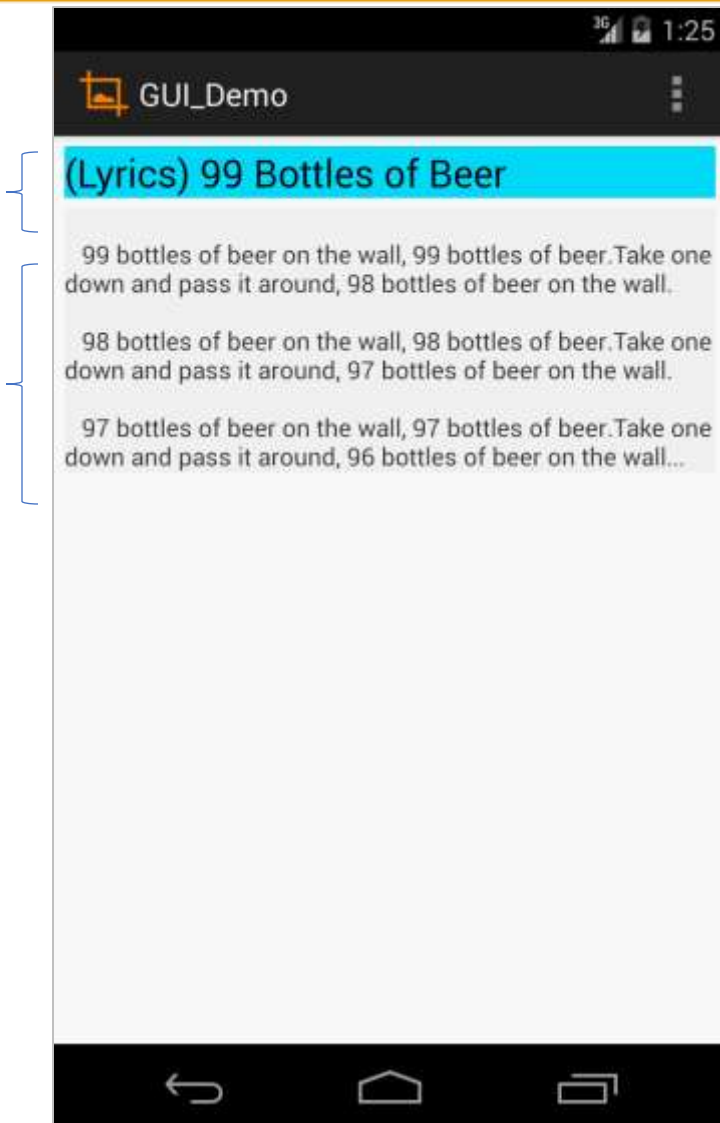
Consider the screen on the right. To make its 'Go' button widget be responsive to the user's pushing of that button, we may add a listener for the **click event**.



```
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



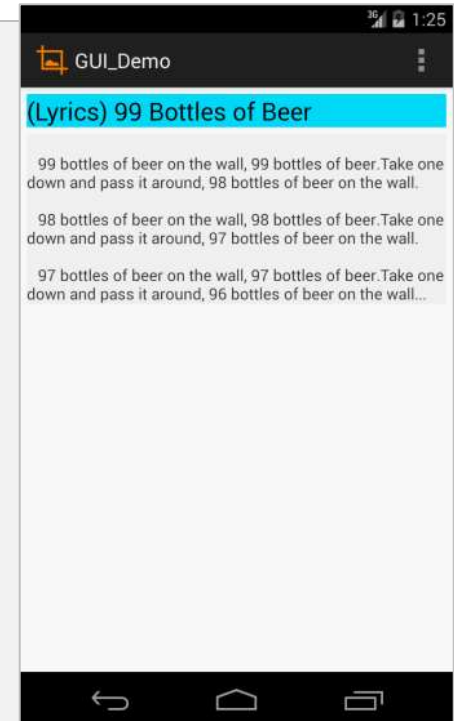
- In Android a **label** or **text-box** is called a **TextView**.
- A TextView is typically used for showing a caption or a text message.
- TextViews are *not* editable, therefore they take no input.
- The text to be shown may include the `\n` formatting character (newLine)
- You may also use HTML formatting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

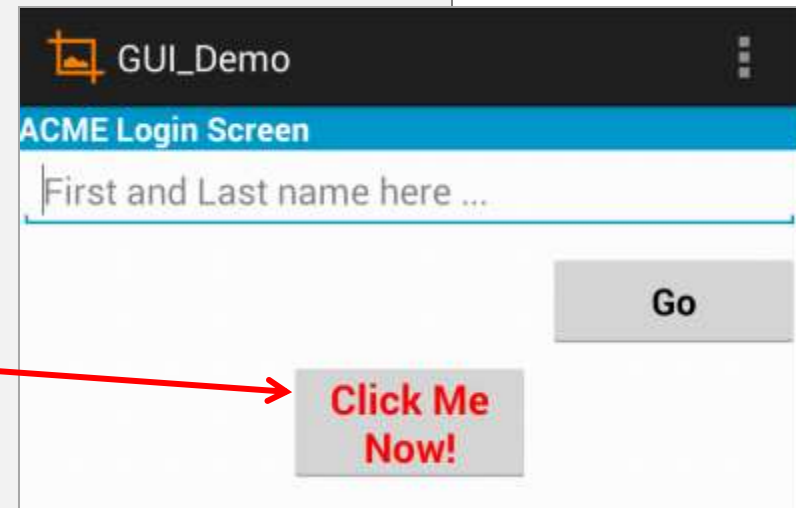
```
val textView: TextView = findViewById(R.id.textView)  
textView.typeface = Typeface.createFromAsset(assets, "fontname.ttf")
```


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.

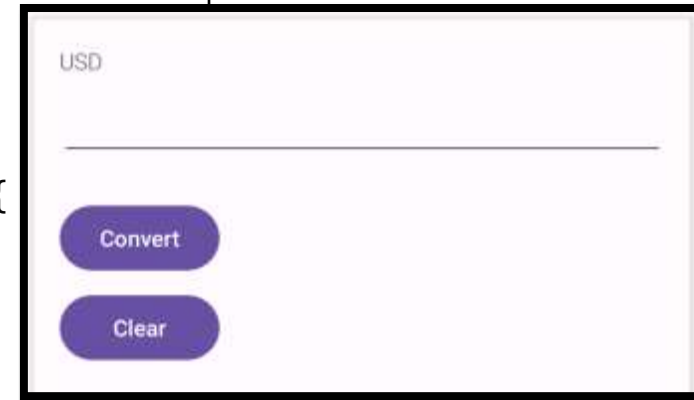
<Button

```
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" />
```



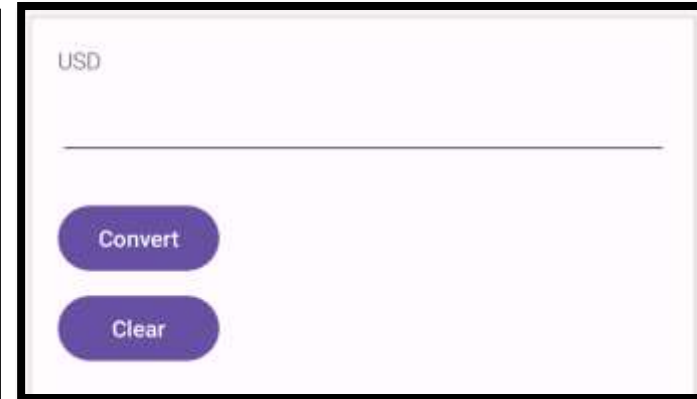
Example: Connecting buttons

```
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  
        }  
    }  
}
```



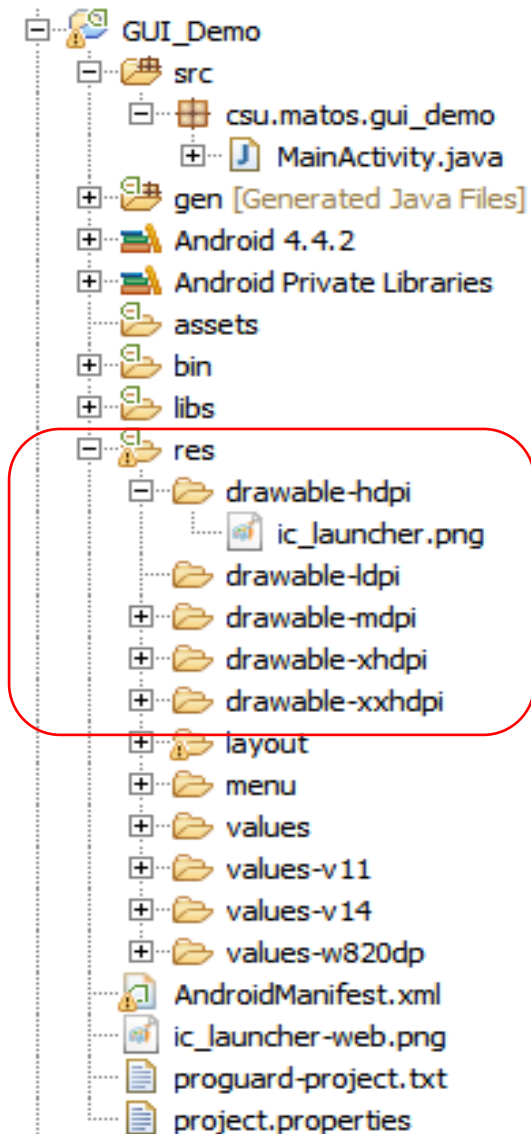
Example: Connecting buttons

```
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" >
```

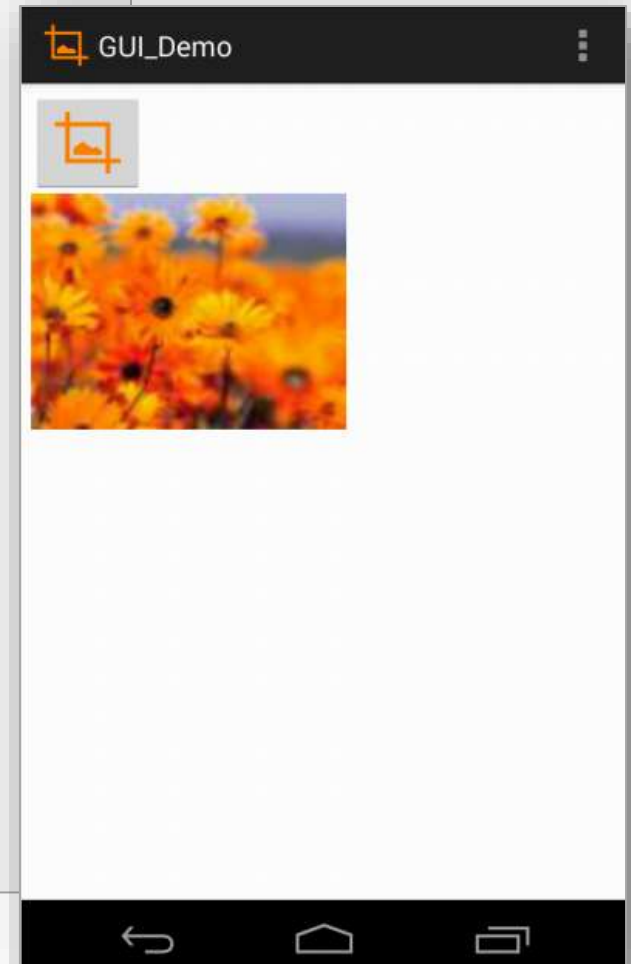
</ImageButton>

<ImageView

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

</ImageView>

</LinearLayout>



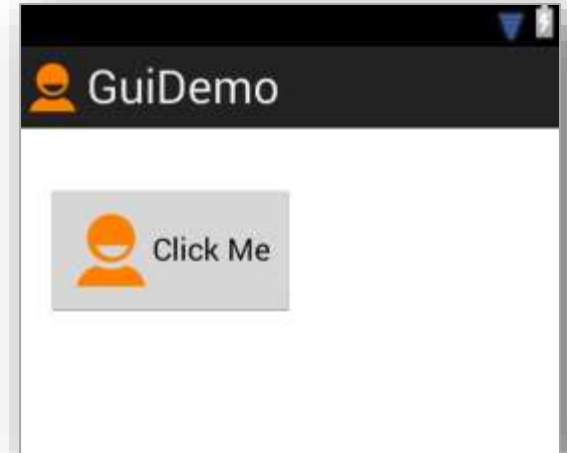
Basic Widgets: Buttons - Combining Images & Text

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



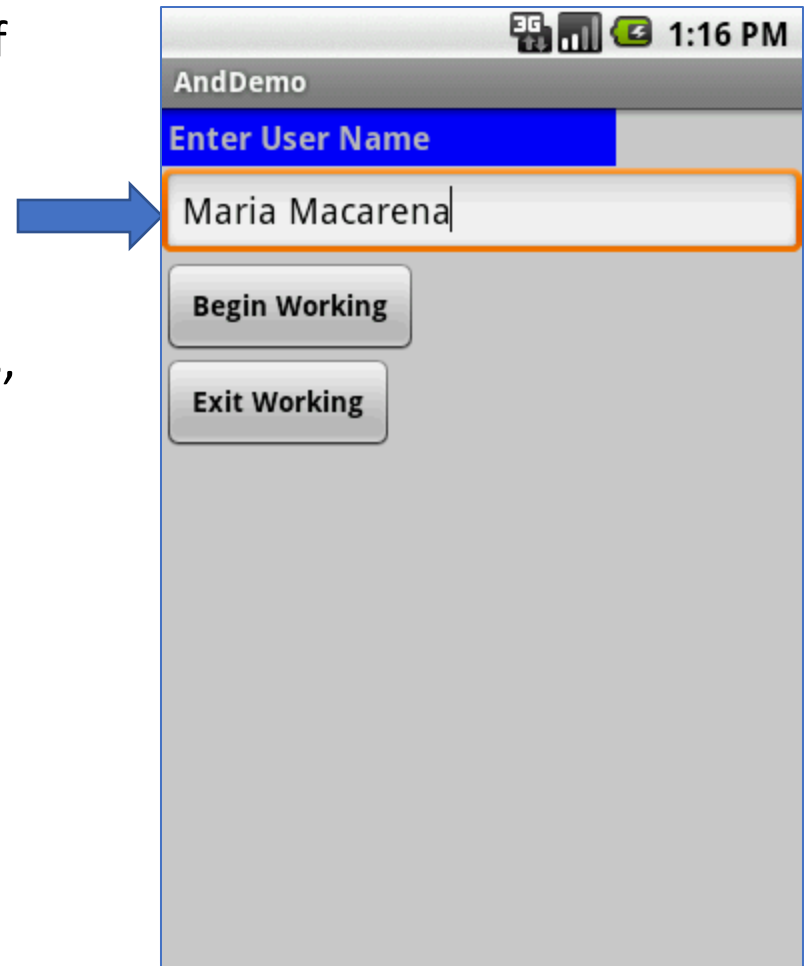
xx-hdpi (2.47KB)
3x = 144 x 144 px

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.

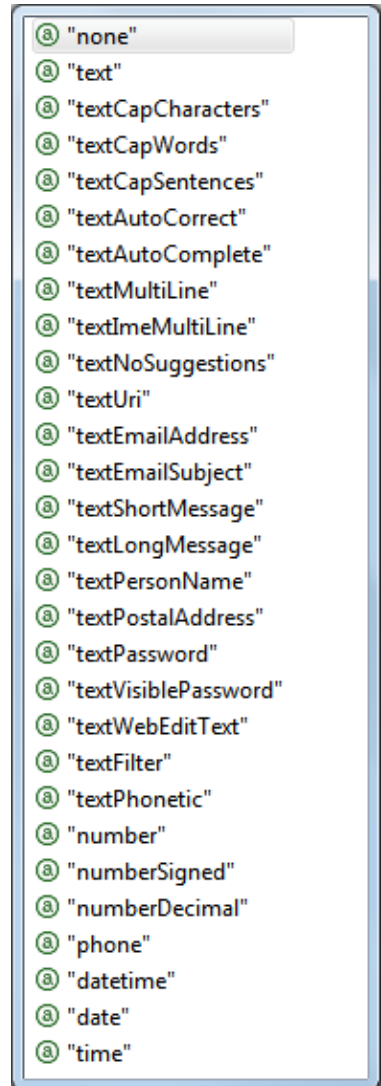
Setting the EditText box to accept a particular choice of data-type, is done through the XML clause

android:inputType="choices"

where **choices** include any of the single values shown in the figure. You may combine types, for instance:

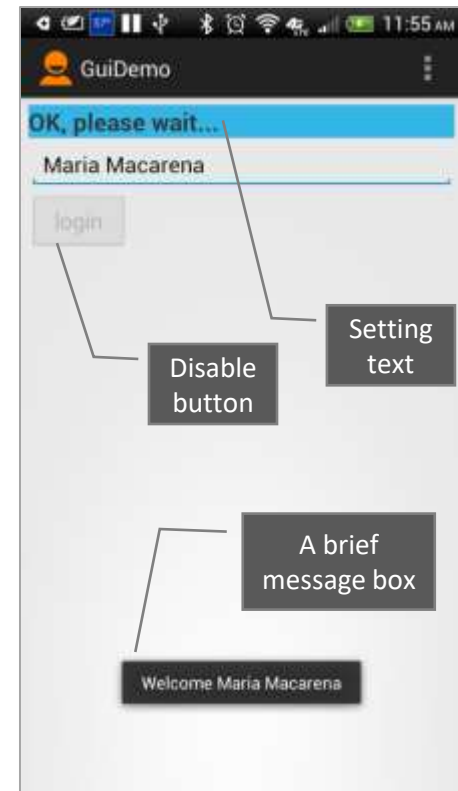
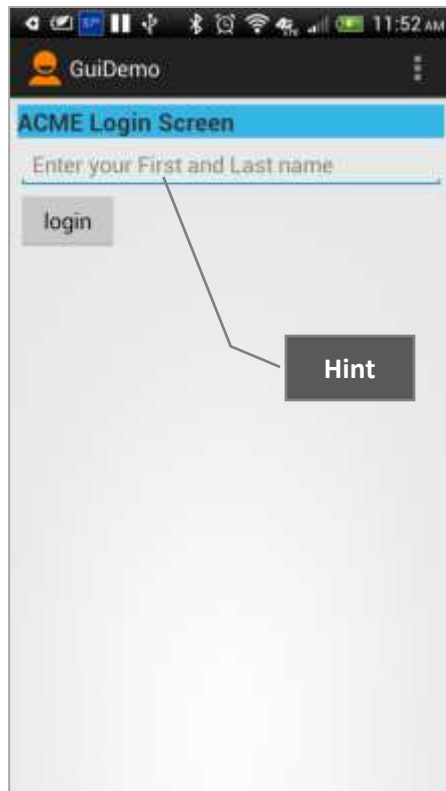
textCapWords | textAutoCorrect

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 (**TextView**), 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")
```

← **Log.v** used for debugging, show in Logcat window

```
            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()  
            }  
        }  
    }  
}
```

Basic Widgets: CheckBoxes

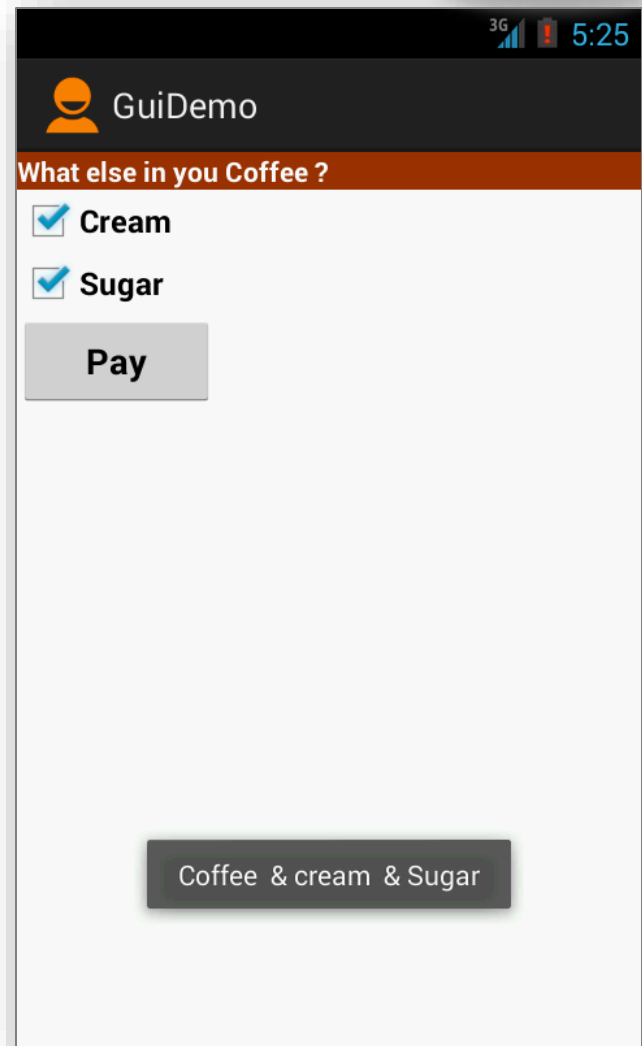


A checkbox is a special **two-states** button which can be either *checked* or *unchecked*.

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.



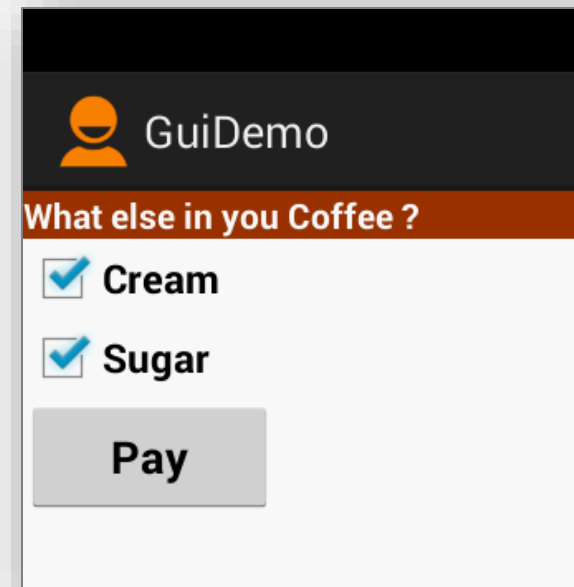
Example: CheckBoxes – CaféApp [Layout 1 of 2]



```
<?xml 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]



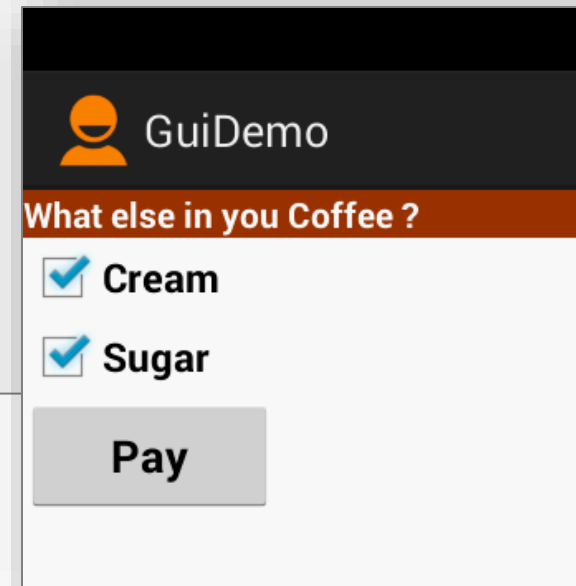
<CheckBox

```
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/...]

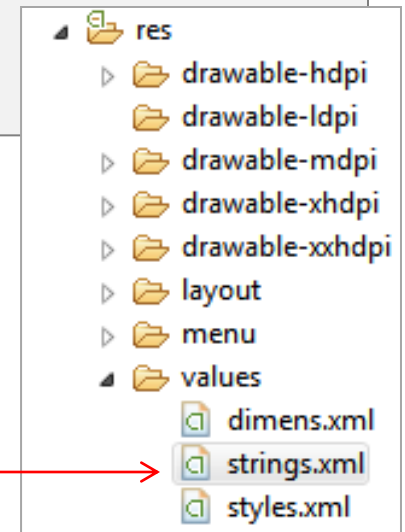


Resources: res/values/strings

```
<?xml 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

The screenshot shows a mobile application interface for a coffee shop. At the top, there's a status bar with '3G', a battery icon, and the time '2:32'. Below that is a header bar with an orange smiley face icon and the text 'GuiDemo'. The main content area has two sections. The first section is titled 'What kind of Coffee?' in a brown header bar. It contains three radio button options: 'Decaf', 'Espresso' (which is selected with a blue dot), and 'Colombian'. The second section is titled 'What else do you like in your coffee?' in a brown header bar. It contains two checkbox options: 'Cream' (unchecked) and 'Sugar' (checked with a blue checkmark). Below these options is a grey button labeled 'Pay'. At the bottom of the screen, there is a dark grey button labeled 'Espresso Coffee & Sugar', which represents the summary of the user's 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/radEspresso"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/espresso" />

    <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 radEspresso: RadioButton = findViewById(R.id.radEspresso)
        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 (radEspresso.isChecked) msg = "Espresso $msg"
            if (radDecaf.isChecked) msg = "Decaf $msg"
        }
    }
}
```

Example: CheckBoxes – CaféApp [MainActivity]



```
// Alternative
when (radioId) {
    R.id.radColombian -> msg = "Colombian $msg"
    R.id.radEspresso -> msg = "Espresso $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`)

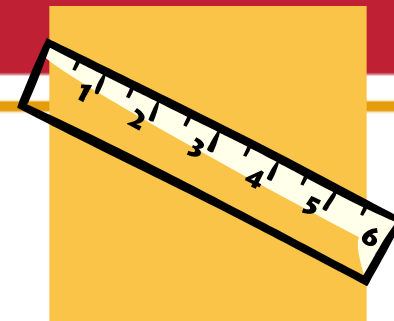
Instead you should apply a two steps procedure in which

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?

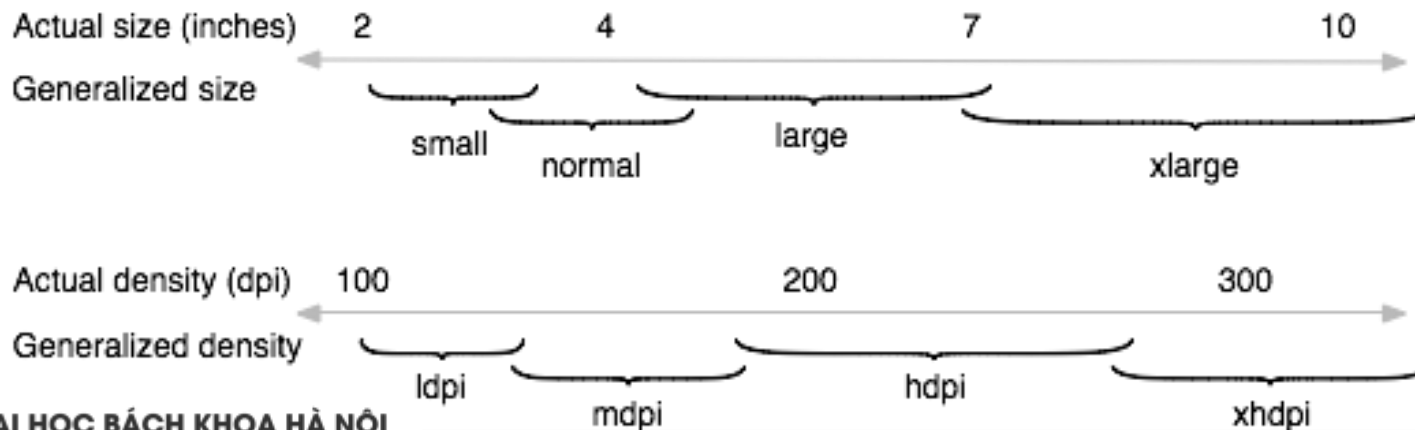
Illustration of how the Android platform maps actual screen densities and sizes to generalized density and size configurations.

A set of four generalized **screen sizes**

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

A set of six generalized **densities**:

<i>ldpi</i>	~120dpi (low)
<i>mdpi</i>	~160dpi (medium)
<i>hdpi</i>	~240dpi (high)
<i>xhdpi</i>	~320dpi (extra-high)
<i>xxhdpi</i>	~480dpi (extra-extra-high)
<i>xxxhdpi</i>	~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: $[2 \times 160] \times [3 \times 160]$)

Assume you want a 120dp button to be placed in the middle of the screen.

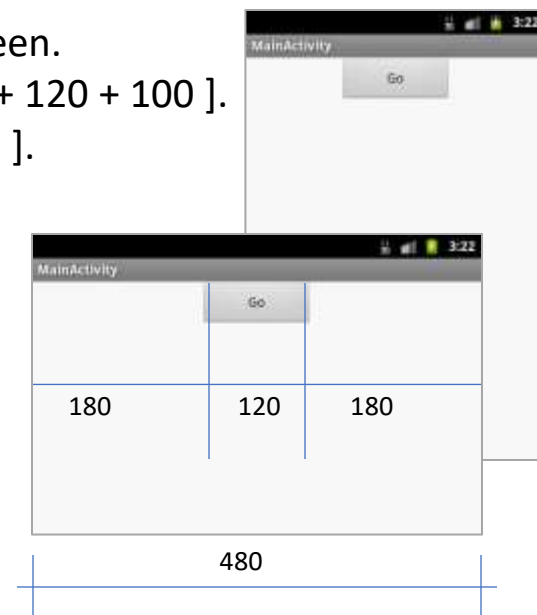
On portrait mode you could allocate the 320 horizontal pixels as $[100 + 120 + 100]$.

On Landscape mode you could allocate 480 pixels as $[180 + 120 + 180]$.

The XML would be

<Button

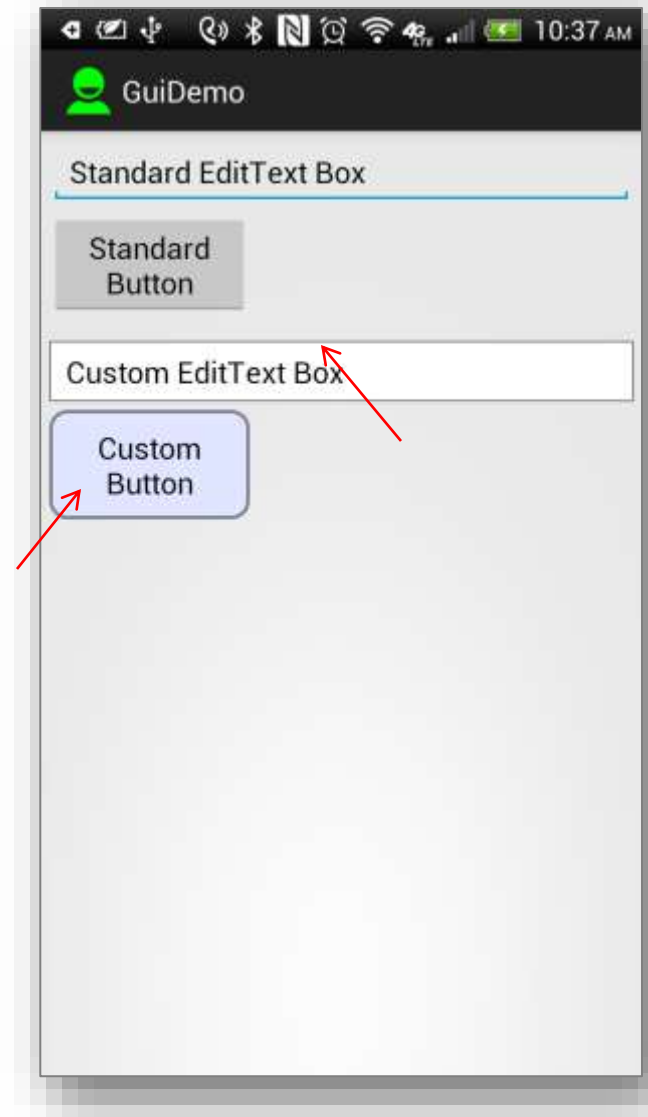
```
android:id="@+id/button1"
android:layout_height="wrap_content"
android:layout_width="120dp"
android:layout_gravity="center"
android:text="@+id/go_caption" />
```



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

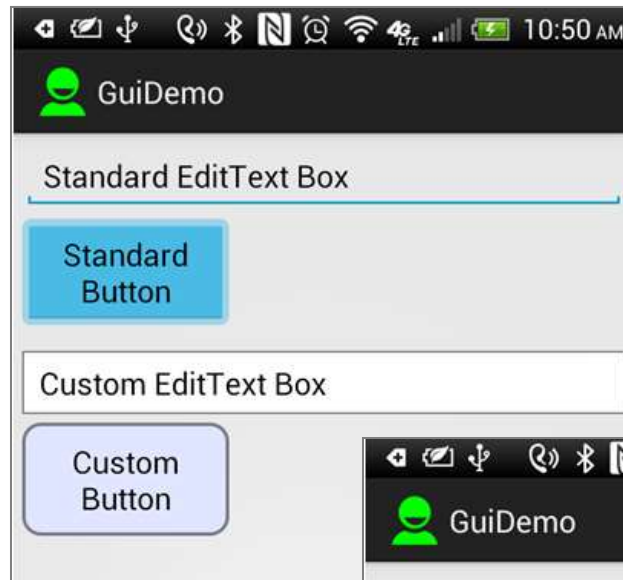
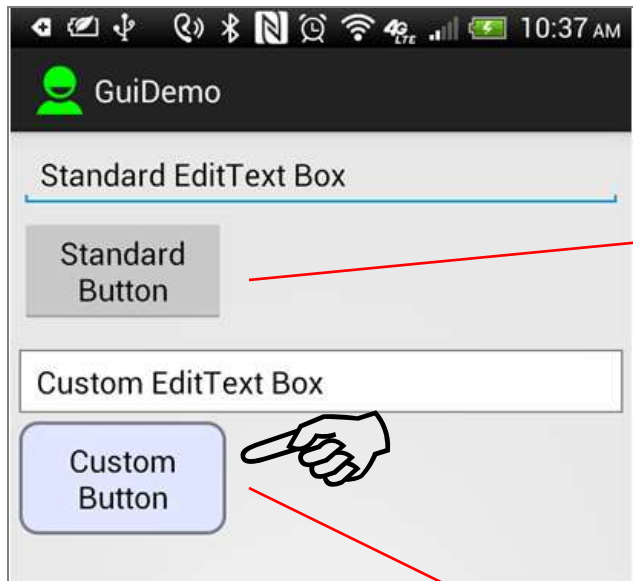
Appendix. Customizing Widgets

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.



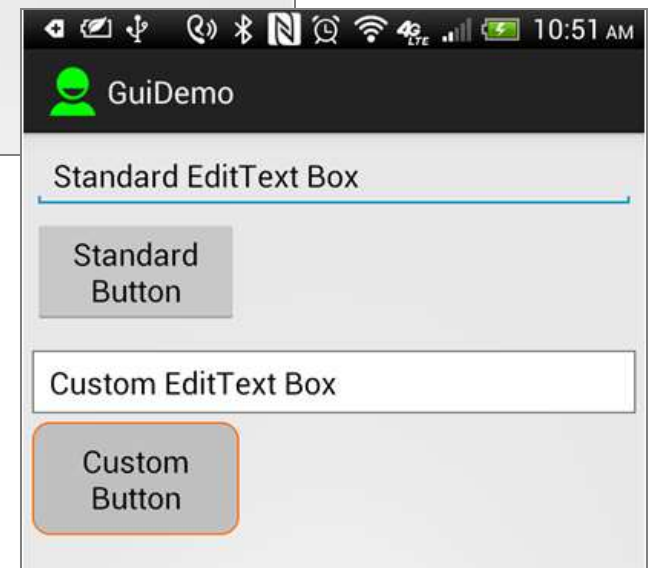
Appendix. Customizing Widgets

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



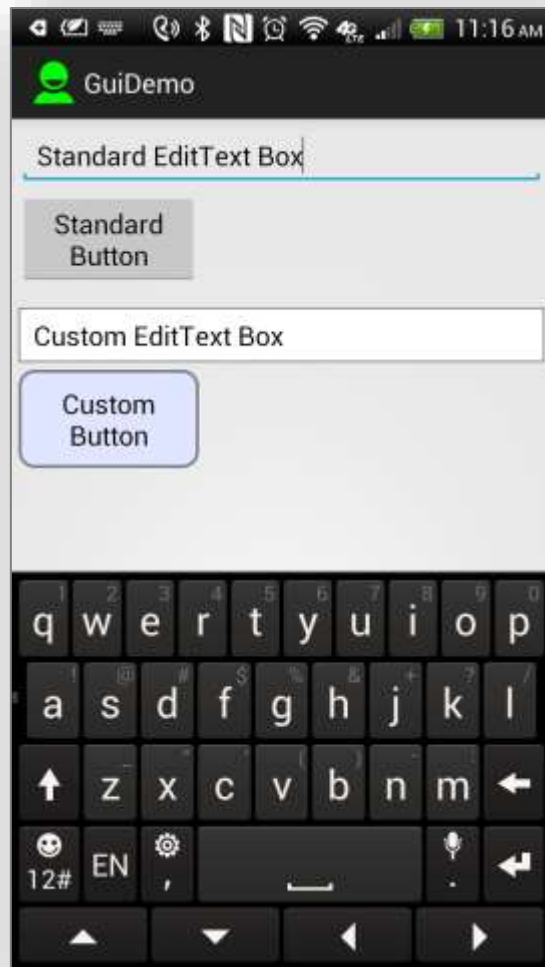
Standard behavior – buttons turns **blue** when it is pressed.

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



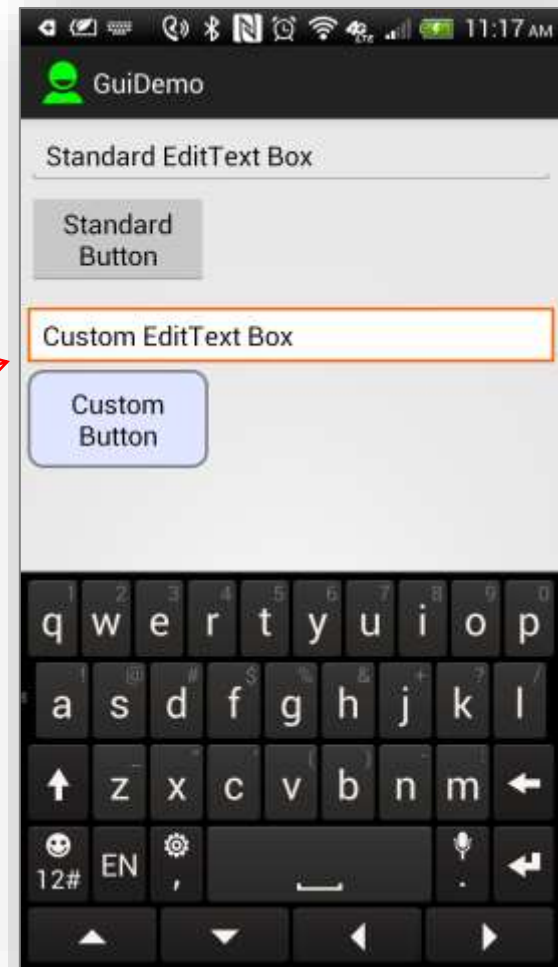
Appendix. Customizing Widgets

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



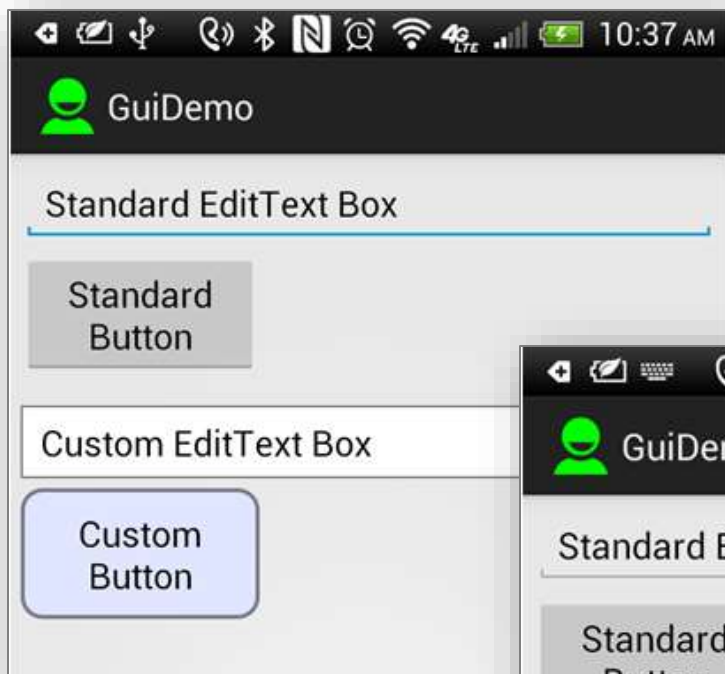
When focused
the standard box
shows a blue
bottom line

A focused
custom box
shows an orange
all-around frame



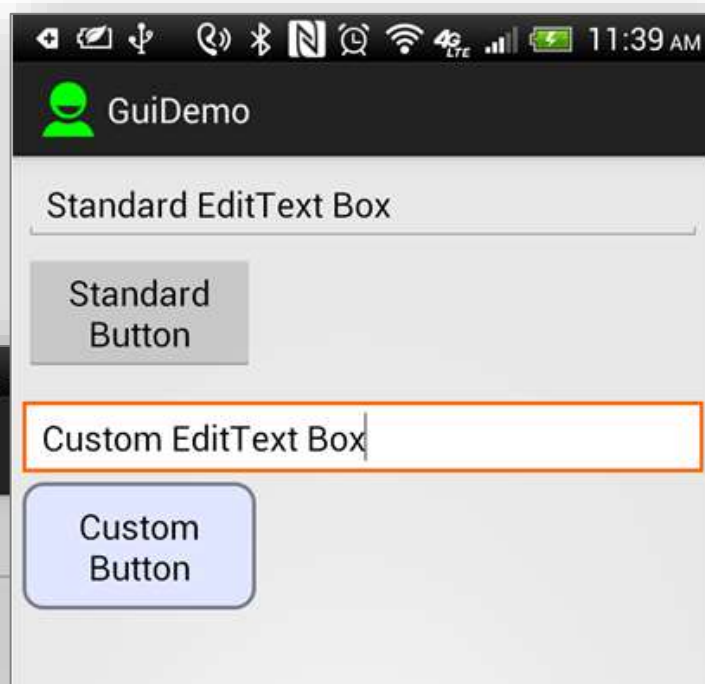
Appendix. Customizing Widgets

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



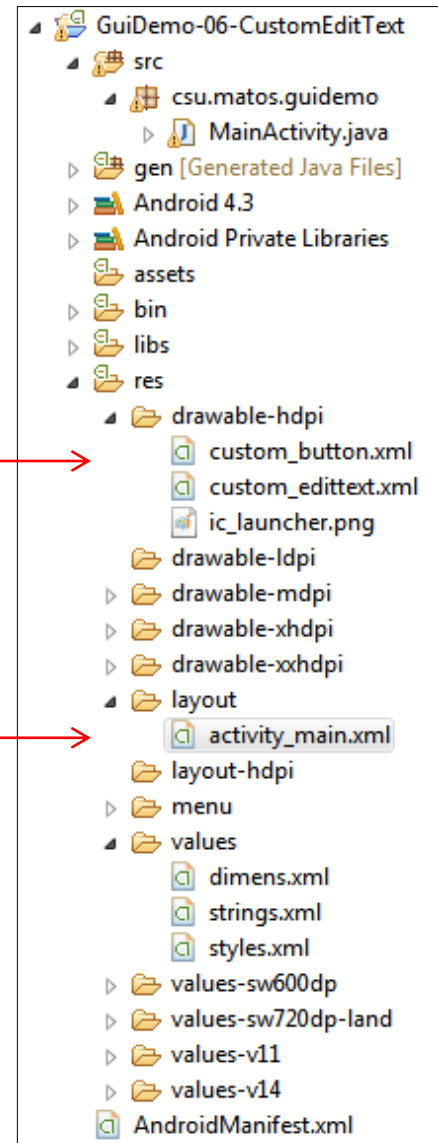
3. Focused custom EditText widget showing an orange border

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

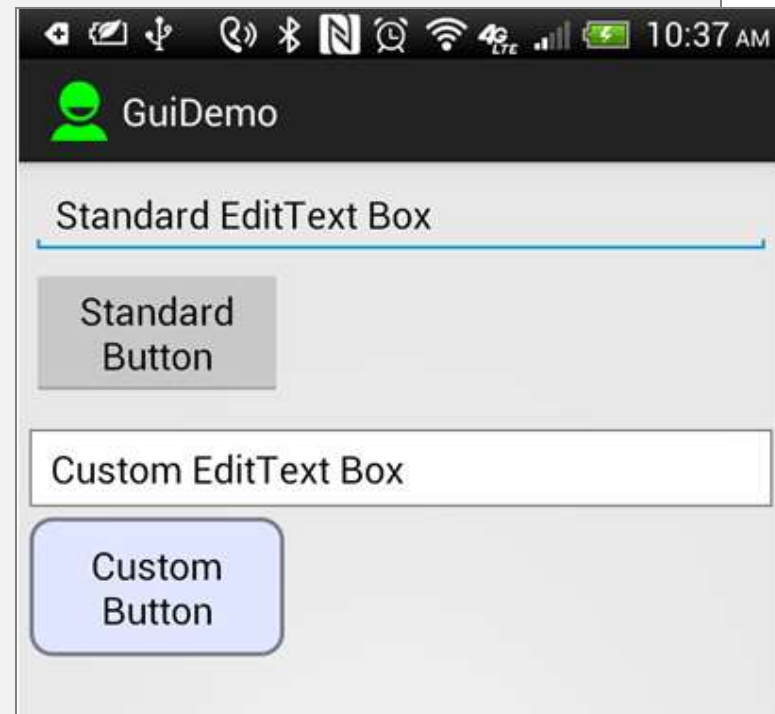
Activity Layout 1 of 2

```
<?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

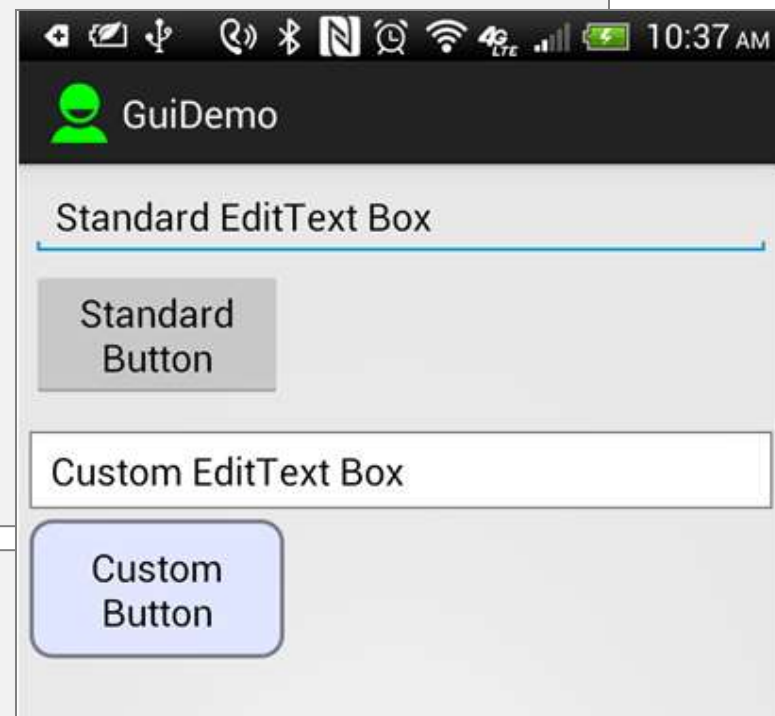
Activity Layout (2 of 2) and Resource: res/values/strings

```
<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>
```

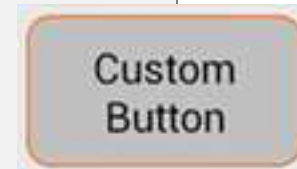


Appendix. Customizing Widgets

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 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 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="#FFFFFF" />
        <stroke android:width="2dp" android:color="#FF6600" />
        <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="#FF7777" />
        <solid android:color="#FFFFFF" />
        <corners android:radius="0dp" />
        <padding android:left="10dp"
            android:top="6dp"
            android:right="10dp"
            android:bottom="6dp" />
    </shape>
</item>
</selector>
```

A rectangular text box with a white background and a thick orange border. The text "Custom EditText Box" is centered inside the box.A rectangular text box with a white background and a thin gray border. The text "Custom EditText Box" is centered inside the box.