Mobile Application Development

Building User Interfaces

(Professional Android 4 Application Development chapter 4)

Outline

- Fundamental Android UI Design, Assigning UI To Activities, Layout Classes, Defining Layouts
- Linear Layout, Relative Layout, Grid Layout, Optimizing Layouts
- Fragments, Creating New Fragment, Fragment LifeCycle, Fragment Manager, Fragment Specific LifeCycle Events
- Fragment States, Adding Fragments to Activities, Using Fragment Transaction
- Interfacing Between Fragments and Activities, Fragment Without User Interfaces, Android Fragment Classes.

FUNDAMENTAL ANDROID UI DESIGN

- Views — Views are the base class for all visual interface elements (commonly known as controls or widgets). All UI controls, including the layout classes, are derived from View.
- View Groups — View Groups are extensions of the View class that can contain multiple child Views. Extend the ViewGroup class to create compound controls made up of interconnected child Views.
- Fragments — Fragments, introduced in Android 3.0 (API level 11), are used to encapsulate portions of your UI. This encapsulation makes Fragments particularly useful when optimizing your UI layouts for different screen sizes and creating reusable UI elements.

FUNDAMENTAL ANDROID UI DESIGN

- Activities — Activities, described in detail in the previous chapter, represent the window, or screen, being displayed. Activities are the Android equivalent of Forms in traditional Windows desktop development. To display a UI, you assign a View (usually a layout or Fragment) to an Activity.

Assigning User Interfaces to Activities

- A new Activity starts with a temptingly empty screen onto which you place your UI. To do so, call setContentView, passing in the View instance, or layout resource, to display.
- The setContentView method accepts either a layout's resource ID or a single View instance.
- @Override
- public void onCreate(Bundle savedInstanceState) {

  super.onCreate(savedInstanceState);

  setContentView(R.layout.main);
- }

Assigning User Interfaces to Activities

- You can obtain a reference to each of the Views within a layout using the findViewById method:
- TextView myTextView = (TextView)findViewById(R.id.textview1);
- If you prefer the more traditional approach, you can construct the UI in code:
- @Override
- public void onCreate(Bundle savedInstanceState) {

  super.onCreate(savedInstanceState);

  TextView myTextView = new TextView(this);

  setContentView(myTextView);

  myTextView.setText("Hello, Android");
- }

LAYOUTS CLASSES

- FrameLayout — The simplest of the Layout Managers, the Frame Layout pins each child view within its frame.
- The default position is the top-left corner, though you can use the gravity attribute to alter its location. Adding multiple children stacks each new child on top of the one before, with each new View potentially obscuring the previous ones.
- LinearLayout — A Linear Layout aligns each child View in either a vertical or a horizontal line. A vertical layout has a column of Views, whereas a horizontal layout has a row of Views.
- The Linear Layout supports a weight attribute for each child View that can control the relative size of each child View within the available space.

LAYOUTS CLASSES

- RelativeLayout — One of the most flexible of the native layouts, the Relative Layout lets you define the positions of each child View relative to the others and to the screen boundaries.
- GridLayout — Introduced in Android 4.0 (API level 14), the Grid Layout uses a rectangular grid of infinitely thin lines to lay out Views in a series of rows and columns.
- The Grid Layout is incredibly flexible and can be used to greatly simplify layouts and reduce or eliminate the complex nesting often required to construct UIs using the layouts described above.

Linear Layout

- The following snippet shows a simple layout that places a TextView above an EditText control using a vertical LinearLayout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Enter Text Below"
  />
  <EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Text Goes Here!"
  />
</LinearLayout>
```


Defining Layouts

- The wrap_content constant sets the size of a View to the minimum required to contain the contents it displays (such as the height required to display a wrapped text string).
- The match_parent constant expands the View to match the available space within the parent View, Fragment, or Activity.
- When preferred, or required, you can implement layouts in code.
- When assigning Views to layouts in code, it's important to apply LayoutParameters using the setLayoutParams method, or by passing them in to the addView call:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);

TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);

myTextView.setText("Enter Text Below");
myEditText.setText("Text Goes Here!");

int lHeight = LinearLayout.LayoutParams.MATCH_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;

ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myEditText, new LinearLayout.LayoutParams(lHeight, lWidth));
setContentView(ll);
```


Linear Layout

- The Linear Layout is one of the simplest layout classes. It allows you to create simple UIs (or UI elements) that align a sequence of child Views in either a vertical or a horizontal line.
- The simplicity of the Linear Layout makes it easy to use but limits its flexibility. In most cases you will use Linear Layouts to construct UI elements that will be nested within other layouts.
- Figure shows two nested Linear Layouts — a horizontal layout of two equally sized buttons within a vertical layout that places the buttons above a List View.

```
<?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">
  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="5dp">
    <Button
      android:text="@string/cancel_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
    <Button
      android:text="@string/ok_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>
  <ListView
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
</LinearLayout>
```


Android - Graphical User

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.

LinearLayout

Setting Attributes

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

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

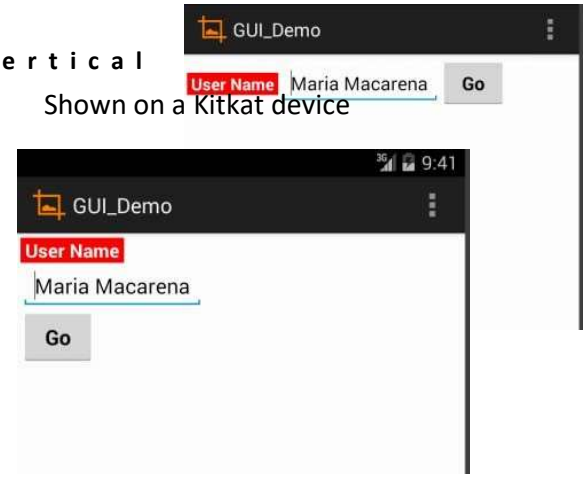
LinearLayout : Orientation

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

horizontal

vertical

Shown on a Kitkat device




```
<LinearLayout  xmlns:android=
"http://schemas.android.com/a
                                                    p
k/res/android"
    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>
```

Android - Graphical User

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).

natural sizes
empty screen space

Shown on a Gingerbread device

Android - Graphical User

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**)

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

1. **match_parent** (previously called ‘**fill_parent**’) indicates the widget wants to be as big as the enclosing parent.

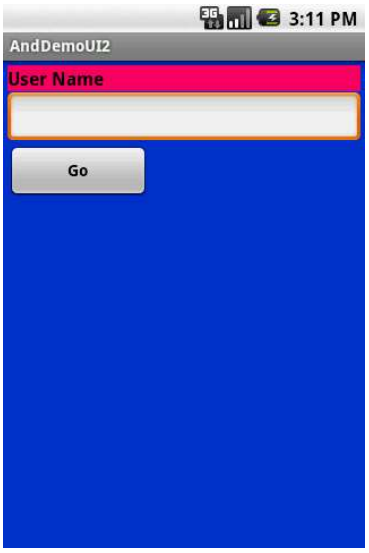
LinearLayout : Fill Model

125 dp

entire row

(320 dp on dium resolution screens)

me



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

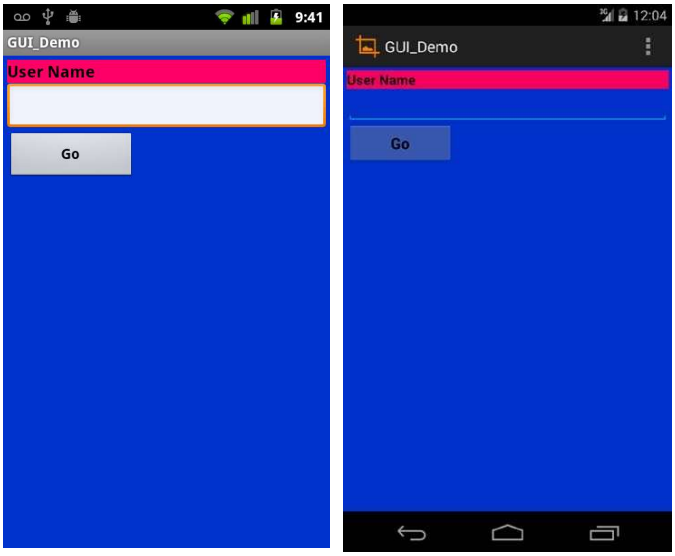
Use all the row

Specific size: 125dp

Warning ! Same XML different rendition…

Since the introduction of Android 4.x, changes in the SDK make layouts to be more *uniformly* displayed in all 4.x and newer devices (the intention is to provide a seamless Android experience independent from provider, hardware, and developer).

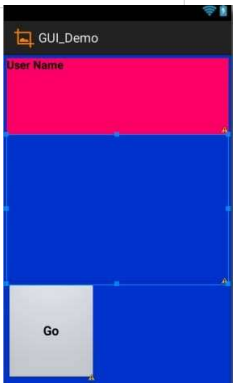
The XML spec used in the previous example *looks* different when displayed on a 4.x and older devices (see figures on the right, please also notice the *color bleeding* occurring on top of the GO button, more on this issue in the Appendix)



Same XML layout shown on a Gingerbread (left) and Kitkat (right) device.

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.



Kitkat

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"
```

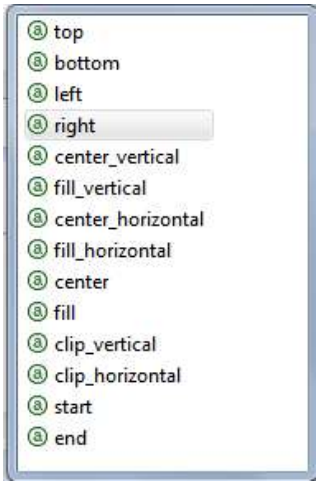
Remember, default value is 0

Gingerbread

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

Android - Graphical User

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.
- Either use
  - `android:padding` property
  - or call method `setPadding()` at runtime.

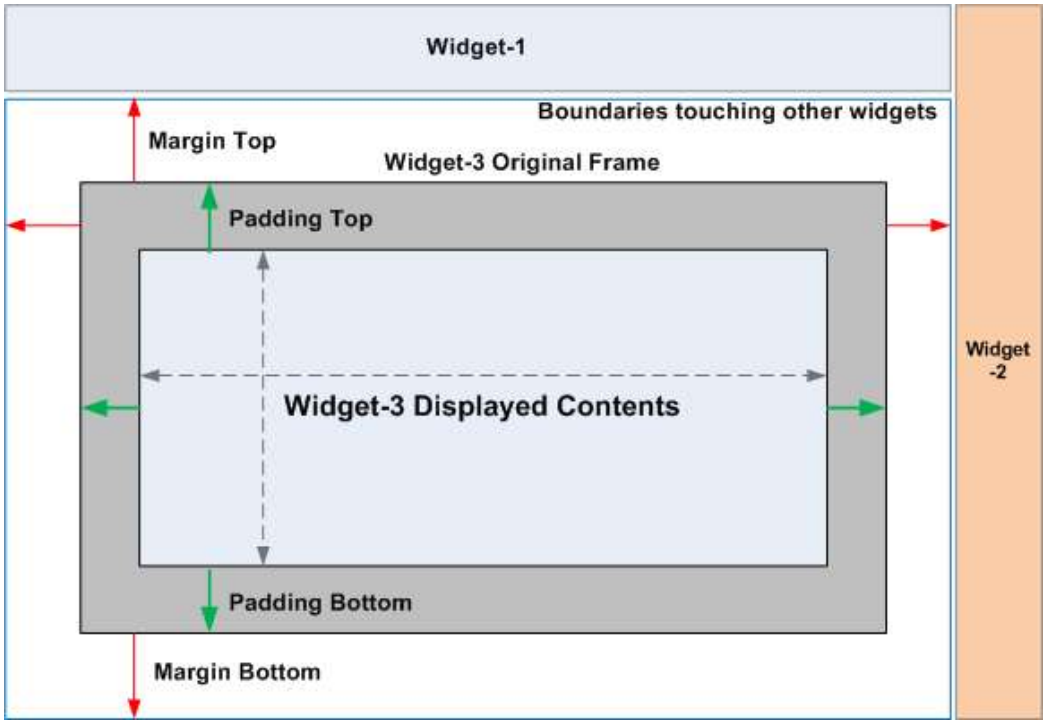
Hello world

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

Android - Graphical User

LinearLayout : Padding and Margin

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



Android - Graphical User

LinearLayout : Set Internal Margins Using Padding

Example:

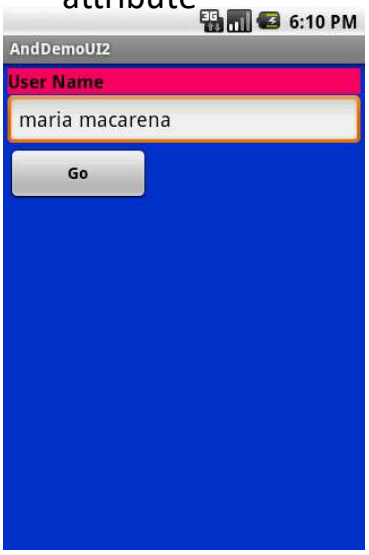
The EditText box has been changed to include 30dp of padding all around


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


Android - Graphical User

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

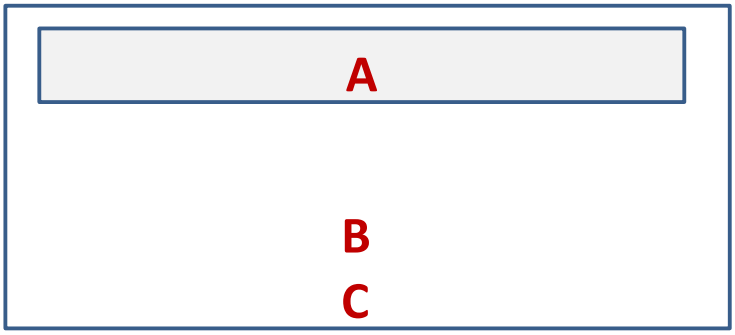
- The Relative Layout provides a great deal of flexibility for your layouts, allowing you to define the position of each element within the layout in terms of its parent and the other Views.


```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <LinearLayout
    android:id="@+id/button_bar"
    android:layout_alignParentBottom="true"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="5dp">
    <Button
      android:text="@string/cancel_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
    <Button
      android:text="@string/ok_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>
  <ListView
    android:layout_above="@id/button_bar"
    android:layout_alignParentLeft="true"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </ListView>
</RelativeLayout>
```

Android - Graphical User

Relative Layout

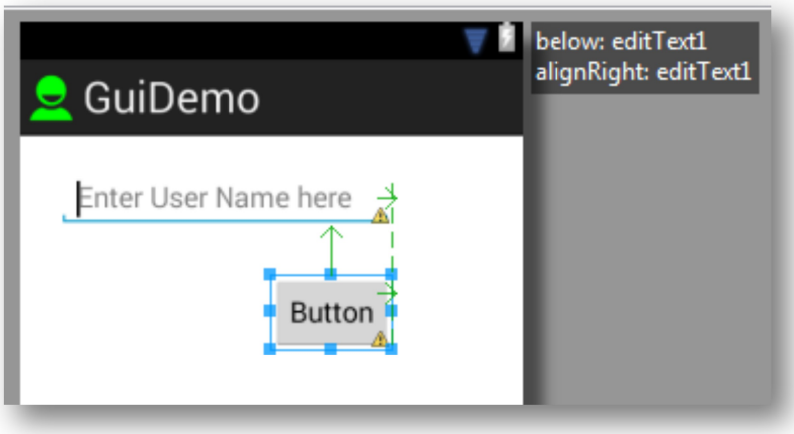
The placement of a widget in a **RelativeLayout** is based on its *positional relationship* to other widgets in the container as well as the parent container.



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

Android - Graphical User

Relative Layout - Example



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

Android - Graphical User

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
```

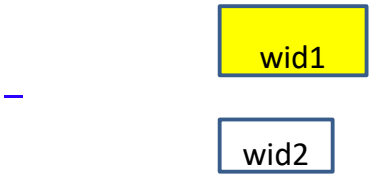

Android - Graphical User

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

- **android:layout_centerHorizontal** the widget should be positioned horizontally at the center of the container
- **android:layout_centerVertical** the widget should be positioned vertically at the center of the container
- **android:layout_centerInParent** the widget should be positioned both horizontally and vertically at the center of the container

- **android:layout_above** indicates that the widget should be placed above the widget referenced in the property
- **android:layout_below** indicates that the widget should be placed below the widget referenced in the property
- **android:layout_toLeftOf** indicates that the widget should be placed to the left of the widget referenced in the property
- **android:layout_toRightOf** indicates that the widget should be placed to the right of the widget referenced in the property

Android - Graphical User

Relative Layout - Referring to Other Widgets

When using relative positioning you need to:

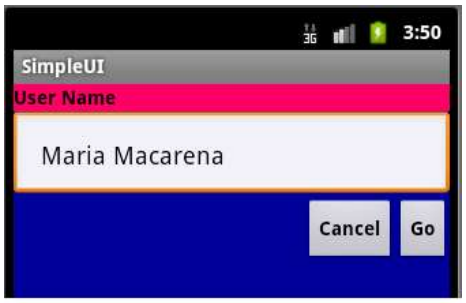
1. Use identifiers (`android:id` attributes) on *all elements* that you will be referring to.

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/r  es/android"
    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>
```



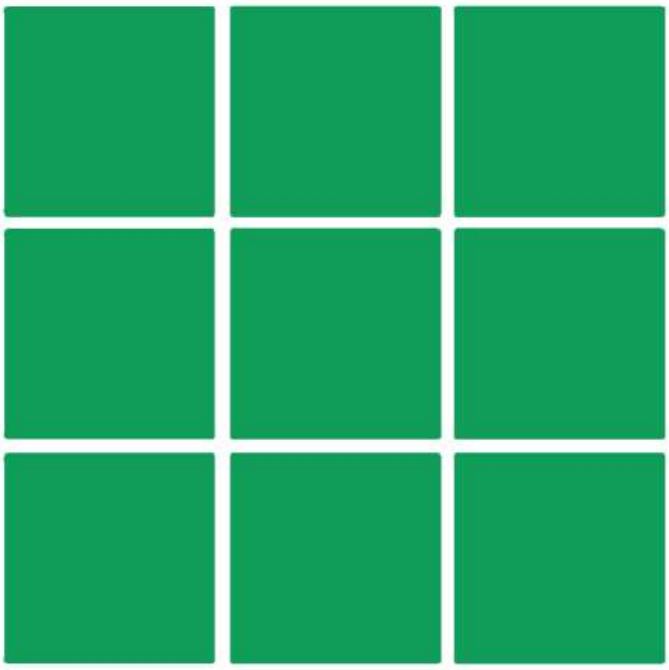

```
    <EditText
        android:id="@+id/txtUserName"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_below="@+id/lblUserName"
        android:padding="20dp" >
    </EditText>

    <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>
```

309

Grid Layout

- The Grid Layout was introduced in Android 3.0 (API level 11) and provides the most flexibility of any of the Layout Managers.
- The Grid Layout is particularly useful for constructing layouts that require alignment in two directions — for example, a form whose rows and columns must be aligned.
- It's also possible to replicate all the functionality provided by the Relative Layout by using the Grid Layout and Linear Layout in combination
- Figure shows the same layout as described in Listing 4-2 using a Grid Layout to replace the Relative Layout.

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">
   <ListView
    android:background="#FF444444"
    android:layout_gravity="fill">
  </ListView>
  <LinearLayout
    android:layout_gravity="fill_horizontal"
    android:orientation="horizontal"
    android:padding="5dp">
    <Button
      android:text="Cancel"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
    <Button
      android:text="OK"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>
</GridLayout>
```

Grid Layout

- XML Attributes of GridView
  - android:numColumns: This attribute of GridView will be used to decide the number of columns that are to be displayed in Grid.
  - android:horizontalSpacing: This attribute is used to define the spacing between two columns of GridView.
  - android:verticalSpacing: This attribute is used to specify the spacing between two rows of GridView.

Frame Layout

- Frame Layout is one of the simplest layout to organize view controls. They are designed to block an area on the screen.
- Frame Layout should be used to hold child view, because it can be difficult to display single views at a specific area on the screen without overlapping each other.
- We can add multiple children to a FrameLayout and control their position by assigning gravity to each child, using the android:layout_gravity attribute..

Frame Layout

- Attributes of Frame Layout:

1. android:id

This is the unique id which identifies the layout in the R.java file.

Below is the id attribute's example with explanation included in which we define the id for Frame Layout.

<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/frameLayout"

android:layout_width="fill_parent"

android:layout_height="fill_parent"/>

Frame Layout

- Attributes of Frame Layout:

2. android:foreground

Foreground defines the drawable to draw over the content and this may be a color value. Possible color values can be in the form of "#rgb", "#argb", "#rrggbb", or "#aarrggbb". This all are different color code model used.

Example: In Below example of foreground we set the green color for foreground of frameLayout so the ImageView and other child views of this layout will not be shown.

Frame Layout

- Attributes of Frame Layout:

<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"

android:id="@+id/framelayout"

android:layout_width="match_parent"

android:layout_height="match_parent"

android:layout_gravity="center"

android:foregroundGravity="fill"

android:foreground="#0f0"><!--foreground color for a FrameLayout-->

Frame Layout

- Attributes of Frame Layout:

3. android:foregroundGravity

This defines the gravity to apply to the foreground drawable. Default value of gravity is fill. We can set values in the form of “top”, ”center_vertical” , ”fill_vertical”, ”center_horizontal”, ”fill_horizontal”, ”center”, ”fill”, ”clip_vertical”, ”clip_horizontal”, ”bottom”, ”left” or ”right” .

It is used to set the gravity of foreground. We can also set multiple values by using “|”. Ex: fill_horizontal|top .Both the fill_horizontal and top gravity are set to framelayout.

Frame Layout

- Attributes of Frame Layout:

4. android:visibility

This determine whether to make the view visible, invisible or gone.

visible – the view is present and also visible

invisible – The view is present but not visible

gone – The view is neither present nor visible

Frame Layout

- Attributes of Frame Layout:

4. android:visibility

This determine whether to make the view visible, invisible or gone.

visible – the view is present and also visible

invisible – The view is present but not visible

gone – The view is neither present nor visible

Optimizing Layouts

- The Grid Layout was introduced in Android 3.0 (API level 11) and provides the most flexibility of any of the Layout Managers.
- Inflating layouts is an expensive process; each additional nested layout and included View directly impacts on the performance and responsiveness of your application.
- To keep your applications smooth and responsive, it's important to keep your layouts as simple as possible and to avoid inflating entirely new layouts for relatively small UI changes

Optimizing Layouts

Redundant Layout Containers Are Redundant

- A Linear Layout within a Frame Layout, both of which are set to MATCH_PARENT, does nothing but add extra time to inflate.
- Look for redundant layouts, particularly if you've been making signifi cant changes to an existing layout or are adding child layouts to an existing layout.
- Layouts can be arbitrarily nested, so it's easy to create complex, deeply nested hierarchies. Although there is no hard limit, it's good practice to restrict nesting to fewer than 10 levels.

Optimizing Layouts

- **Avoid Using Excessive Views**
- Each additional View takes time and resources to inflate. To maximize the speed and responsiveness of your application, none of its layouts should include more than 80 Views.
- When you exceed this limit, the time taken to inflate the layout becomes significant.
- To minimize the number of Views infl ated within a complex layout, you can use a ViewStub.
- A View Stub works like a lazy include — a stub that represents the specified child Views within the parent layout — but the stub is only inflated explicitly via the inflate method or when it's made visible.

Optimizing Layouts

```
// Find the stub
View stub = findViewById(R.id. download_progress_panel_stub);
// Make it visible, causing it to inflate the child layout
stub.setVisibility(View.VISIBLE);
```

As a result, the Views contained within the child layout aren't created until they are required — minimizing the time and resource cost of inflating complex UIs.

Optimizing Layouts

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout "xmlns:android=http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <ListView
    android:id="@+id/myListView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />
  <ViewStub
    android:id="@+id/download_progress_panel_stub"

    android:layout="@layout/progress_overlay_panel"
    android:inflatedId="@+id/download_progress_panel"

    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom"
  />
</FrameLayout>
```

When adding a View Stub to your layout, you can override the id and layout parameters of the root View of the layout it represents.