Android
Declaring Layout

Key classes

1. View
2. ViewGroup
3. ViewGroup.LayoutParams

In this document

Your layout is the architecture for the user interface in an Activity. It defines the layout structure and holds all the elements that appear to the user. You can declare your layout in two ways:

- **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.
- **Instantiate layout elements at runtime**. Your application can create View and ViewGroup objects (and manipulate their properties) programmatically.

The Android framework gives you the flexibility to use either or both of these methods for declaring and managing your application's UI. For example, you could declare your application's default layouts in XML, including the screen elements that will appear in them and their properties. You could then add code in your application that would modify the state of the screen objects, including those declared in XML, at run time.

The Android Development Tools (ADT) plugin for Eclipse offers a layout preview of your XML — with the XML file opened, select the **Layout** tab.

You should also try the Hierarchy Viewer tool, for debugging layouts — it reveals layout property values, draws wireframes with padding/margin indicators, and full rendered views while you debug on the emulator or device.

The advantage to declaring your UI in XML is that it enables you to better separate the presentation of your application from the code that controls its behavior. Your UI descriptions are external to your application code, which means that you can modify or adapt it without having to modify your source code and recompile. For example, you can create XML layouts for different screen orientations, different device screen sizes, and different languages. Additionally,

declaring the layout in XML makes it easier to visualize the structure of your UI, so it's easier to debug problems. As such, this document focuses on teaching you how to declare your layout in XML. If you're interested in instantiating View objects at runtime, refer to the ViewGroup and View class references.

In general, the XML vocabulary for declaring UI elements closely follows the structure and naming of the classes and methods, where element names correspond to class names and attribute names correspond to methods. In fact, the correspondence is often so direct that you can guess what XML attribute corresponds to a class method, or guess what class corresponds to a given xml element. However, note that not all vocabulary is identical. In some cases, there are slight naming differences. For example, the EditText element has a `text` attribute that corresponds to `EditText.setText()`.

Tip: Learn more about different layout types in Common Layout Objects. There are also a collection of tutorials on building various layouts in the Hello Views tutorial guide.

Write the XML

For your convenience, the API reference documentation for UI related classes lists the available XML attributes that correspond to the class methods, including inherited attributes.

To learn more about the available XML elements and attributes, as well as the format of the XML file, see Layout Resources.

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical LinearLayout to hold a TextView and a Button:

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

After you've declared your layout in XML, save the file with the `.xml` extension, in your Android project's `res/layout/` directory, so it will properly compile.

We'll discuss each of the attributes shown here a little later.

Load the XML Resource

When you compile your application, each XML layout file is compiled into a View resource. You should load the layout resource from your application code, in your Activity.onCreate() callback implementation.

Do so by calling `setContentView()`, passing it the reference to your layout resource in the form of: `R.layout.`*`layout_file_name`* For example, if your XML layout is saved as `main_layout.xml`, you would load it for your Activity like so:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView.(R.layout.main_layout);
}
```

The `onCreate()` callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion on Lifecycles, in the Application Fundamantals, for more on this).

Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the `textSize` attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the `id` attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the application is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the `id` attribute. This is an XML attribute common to all View objects (defined by the View class) and you will use it very often. The syntax for an ID, inside an XML tag is:

```
android:id="@+id/my_button"
```

The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the `R.java` file). There are a number of other ID resources that are offered by the Android framework.

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

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

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

In order to create views and reference them from the application, a common pattern is to:

1. Define a view/widget in the layout file and assign it a unique ID:

```
<Button android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/my_button_text"/>
```

6. Then create an instance of the view object and capture it from the layout (typically in the `onCreate()` method):

```
Button myButton = (Button) findViewById(R.id.my_button);
```

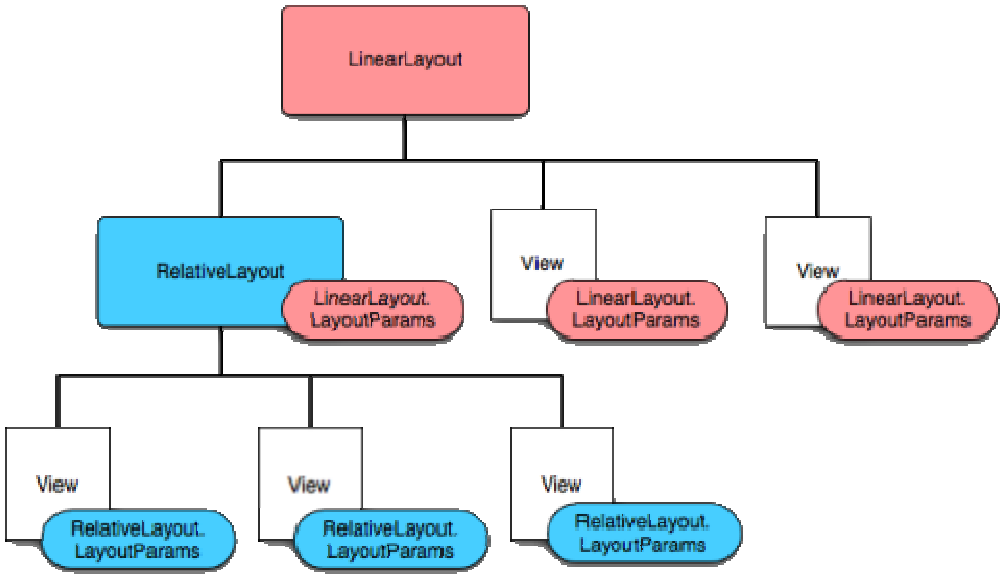
Defining IDs for view objects is important when creating a RelativeLayout. In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

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

Layout Parameters

XML layout attributes named `layout_`*`something`* define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every ViewGroup class implements a nested class that extends ViewGroup.LayoutParams. This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in the figure below, the parent view group defines layout parameters for each child view (including the child view group).



Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

All view groups include a width and height (`layout_width` and `layout_height`), and each view is required to define them. Many LayoutParams also include optional margins and borders. You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will tell your view to size itself either to the dimensions required by its content, or to become as big as its parent view group will allow (with the *wrap_content* and

fill_parent values, respectively). The accepted measurement types are defined in the Available Resources document.

Layout Position

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

It is possible to retrieve the location of a view by invoking the methods getLeft() and getTop(). The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when getLeft() returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, several convenience methods are offered to avoid unnecessary computations, namely getRight() and getBottom(). These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling getRight() is similar to the following computation: `getLeft() + getWidth()`.

Size, Padding and Margins

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

The first pair is known as *measured width* and *measured height*. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling getMeasuredWidth() and getMeasuredHeight().

The second pair is simply known as *width* and *height*, or sometimes *drawing width* and *drawing height*. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling getWidth() and getHeight().

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific amount of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the setPadding(int, int, int, int) method and queried by calling getPaddingLeft(), getPaddingTop(), getPaddingRight() and getPaddingBottom().

Even though a view can define a padding, it does not provide any support for margins. However, view groups provide such a support. Refer to ViewGroup and ViewGroup.MarginLayoutParams for further information.

Common Layout Objects

In this document

This section describes some of the more common types of layout objects to use in your applications. Like all layouts, they are subclasses of ViewGroup.

Also see the Hello Views tutorials for some guidance on using more Android View layouts.

FrameLayout

FrameLayout is the simplest type of layout object. It's basically a blank space on your screen that you can later fill with a single object —

for example, a picture that you'll swap in and out. All child elements of the FrameLayout are pinned to the top left corner of the screen; you cannot specify a different location for a child view. Subsequent child views will simply be drawn over previous ones, partially or totally obscuring them (unless the newer object is transparent).

LinearLayout

LinearLayout aligns all children in a single direction — vertically or horizontally, depending on how you define the `orientation` attribute.

All children are stacked one after the other, so a vertical list will only have one child per row, no matter how wide they are, and a horizontal list will only be one row high (the height of the tallest child, plus padding). A LinearLayout respects *margin*s between children and the *gravity* (right, center, or left alignment) of each child.

LinearLayout also supports assigning a *weight* to individual children. This attribute assigns an "importance" value to a view, and allows it to expand to fill any remaining space in the parent view.

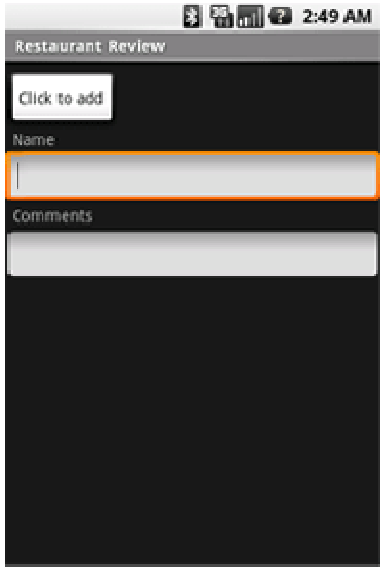
Child views can specify an integer weight value, and then any remaining space in the view group is assigned to children in the proportion of their declared weight. Default weight is zero. For example, if there are three text boxes and two of them declare a weight of 1, while the other is given no weight (0), the third text box without weight will not grow and will only occupy the

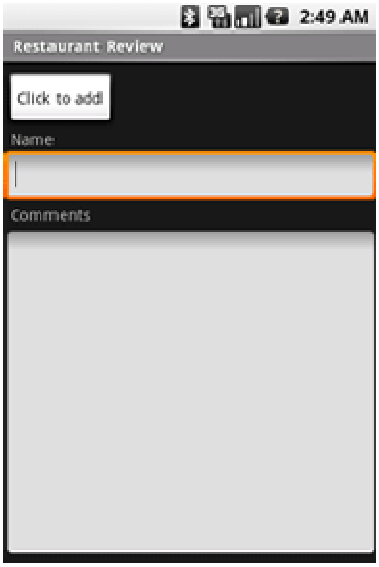
area required by its content. The other two will expand equally to fill the space remaining after all three boxes are measured. If the third box is then given a weight of 2 (instead of 0), then it is now declared "more important" than both the others, so it gets half the total remaining space, while the first two share the rest equally.

Tip: To create a proportionate size layout on the screen, create a container view group object with the `layout_width` and `layout_height` attributes set to *fill_parent*; assign the children `height` or `width` to `0` (zero); then assign relative `weight` values to each child, depending on what proportion of the screen each should have.

The following two forms represent a LinearLayout with a set of elements: a button, some labels and text boxes. The text boxes have their width set to *fill_parent*; other elements are set to *wrap_content*. The gravity, by default, is left.

The difference between the two versions of the form is that the form on the left has weight values unset (0 by default), while the form on the right has the comments text box weight set to 1. If the Name textbox had also been set to 1, the Name and Comments text boxes would be the same height.





Within a horizontal LinearLayout, items are aligned by the position of their text base line (the first line of the first list element — topmost or leftmost — is considered the reference line). This is so that people scanning elements in a form shouldn't have to jump up and down to read element text in neighboring elements. This can be turned off by setting `android:baselineAligned="false"` in the layout XML.

To view other sample code, see the Hello LinearLayout tutorial.

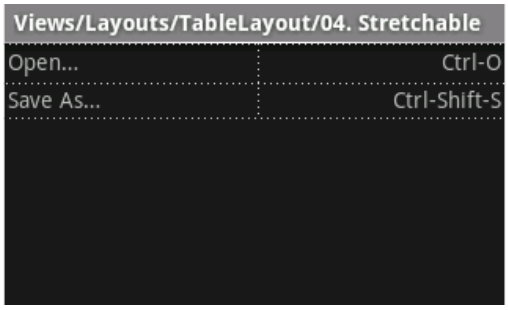
TableLayout

TableLayout positions its children into rows and columns. TableLayout containers do not display border lines for their rows, columns, or cells. The table will have as many columns as the row with the most cells. A table can leave cells empty, but cells cannot span columns, as they can in HTML.

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

The following sample layout has two rows and two cells in each. The accompanying screenshot shows the result, with cell borders displayed as dotted lines (added for visual effect).

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">
    <TableRow>
        <TextView
            android:text="@string/table_layout_4_open"
            android:padding="3dip" />
        <TextView
            android:text="@string/table_layout_4_open_shortcut"
            android:gravity="right"
            android:padding="3dip" />
    </TableRow>

    <TableRow>
        <TextView
            android:text="@string/table_layout_4_save"
            android:padding="3dip" />
        <TextView
            android:text="@string/table_layout_4_save_shortcut"
            android:gravity="right"
            android:padding="3dip" />
    </TableRow>
</TableLayout>
```



Columns can be hidden, marked to stretch and fill the available screen space, or can be marked as shrinkable to force the column to shrink until the table fits the screen. See the TableLayout reference documentation for more details.

RelativeLayout

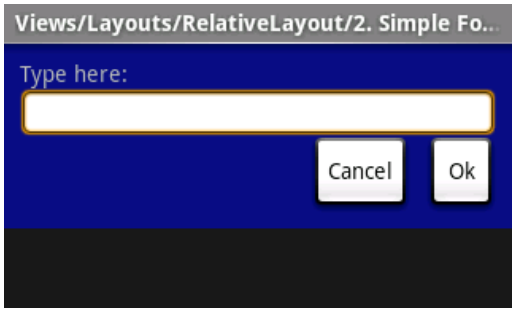
RelativeLayout lets child views specify their position relative to the parent view or to each other (specified by ID).

So you can align two elements by right border, or make one below another, centered in the screen, centered left, and so on. Elements are rendered in the order given, so if the first element is centered in the screen, other elements aligning themselves to that element will be aligned relative to screen center. Also, because of this ordering, if using XML to specify this layout, the element that you will reference (in order to position other view objects) must be listed in the XML file before you refer to it from the other views via its reference ID.

The example below shows an XML file and the resulting screen in the UI. Note that the attributes that refer to relative elements (e.g., *layout_toLeft*) refer to the ID using the syntax of a relative resource (*@id/id*).

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:background="@drawable/blue"
                android:padding="10px" >

    <TextView android:id="@+id/label"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:text="Type here:" />

    <EditText android:id="@+id/entry"
              android:layout_width="fill_parent"
              android:layout_height="wrap_content"
              android:background="@android:drawable/editbox_background"
              android:layout_below="@id/label" />

    <Button android:id="@+id/ok"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@id/entry"
            android:layout_alignParentRight="true"
            android:layout_marginLeft="10px"
            android:text="OK" />

    <Button android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_toLeftOf="@id/ok"
            android:layout_alignTop="@id/ok"
```

```
            android:text="Cancel" />
</RelativeLayout>
```



Some of these properties are supported directly by the element, and some are supported by its LayoutParams member (subclass RelativeLayout for all the elements in this screen, because all elements are children of a RelativeLayout parent object). The defined RelativeLayout parameters are: `width`, `height`, `below`, `alignTop`, `toLeft`, `padding[Bottom|Left|Right|Top]`, and `margin[Bottom|Left|Right|Top]`. Note that some of these parameters specifically support relative layout positions — their values must be the ID of the element to which you'd like this view laid relative. For example, assigning the parameter `toLeft="my_button"` to a TextView would place the TextView to the left of the View with the ID *my_button* (which must be written in the XML *before* the TextView).

Summary of Important View Groups

These objects all hold child UI elements. Some provide their own form of a visible UI, while others are invisible structures that only manage the layout of their child views.

Class	Description
FrameLayout	Layout that acts as a view frame to display a single object.
Gallery	A horizontal scrolling display of images, from a bound list.
GridView	Displays a scrolling grid of m columns and n rows.
LinearLayout	A layout that organizes its children into a single horizontal or vertical row. It creates a scrollbar if the length of the window exceeds the length of the screen.
ListView	Displays a scrolling single column list.
RelativeLayout	Enables you to specify the location of child objects relative to each other (child A to the left of child B) or to the parent (aligned to the top of the parent).
ScrollView	A vertically scrolling column of elements.
Spinner	Displays a single item at a time from a bound list, inside a one-row textbox. Rather like a one-row listbox that can scroll either horizontally or vertically.
SurfaceView	Provides direct access to a dedicated drawing surface. It can hold child views layered on top of the surface, but is intended for applications that need to draw pixels, rather than using widgets.

TabHost	Provides a tab selection list that monitors clicks and enables the application to change the screen whenever a tab is clicked.
TableLayout	A tabular layout with an arbitrary number of rows and columns, each cell holding the widget of your choice. The rows resize to fit the largest column. The cell borders are not visible.
ViewFlipper	A list that displays one item at a time, inside a one-row textbox. It can be set to swap items at timed intervals, like a slide show.
ViewSwitcher	Same as ViewFlipper.

Hello, Views

http://developer.android.com/guide/tutorials/views/index.html

This collection of "Hello World"-style tutorials is designed to get you quickly started with common Android Views and widgets. The aim is to let you copy and paste these kinds of boring bits so you can focus on developing the code that makes your Android application rock. Of course, we'll discuss some of the given code so that it all makes sense.

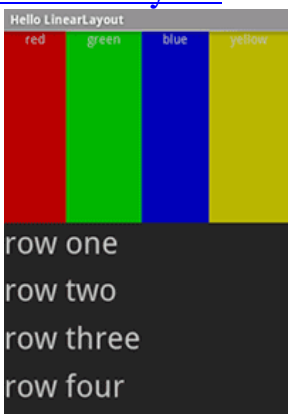
Note that a certain amount of knowledge is assumed for these tutorials. If you haven't completed the Hello, World tutorial, please do so—it will teach you many things you should know about basic Android development and Eclipse features. More specifically, you should know:

- How to create a new Android project.
- The basic structure of an Android project (resource files, layout files, etc.).
- The essential components of an Activity.
- How to build and run a project.

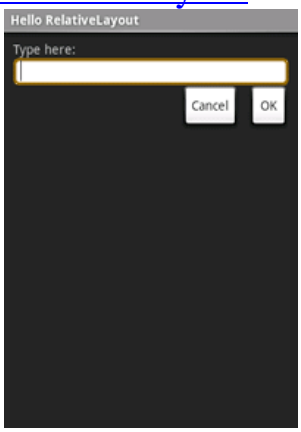
Please, also notice that, in order to make these tutorials simple, some may not convey the better Android coding practices. In particular, many of them use hard-coded strings in the layout files—the better practice is to reference strings from your strings.xml file.

With this knowledge, you're ready to begin, so take your pick.

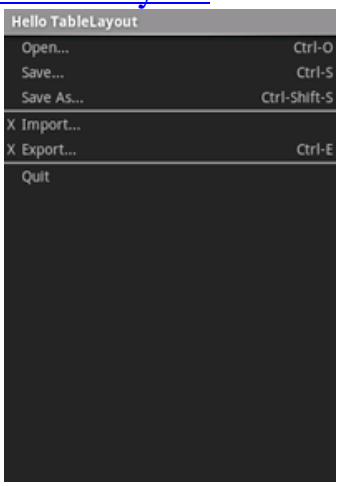
LinearLayout



RelativeLayout



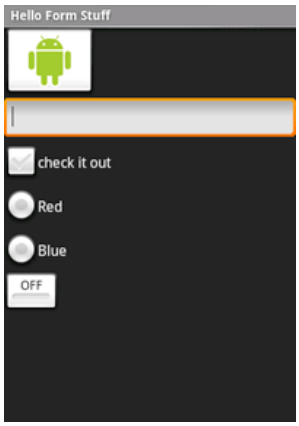
TableLayout



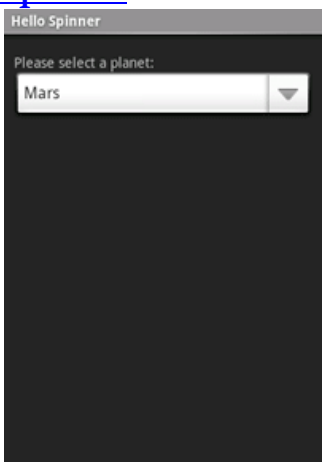
DatePicker

TimePicker

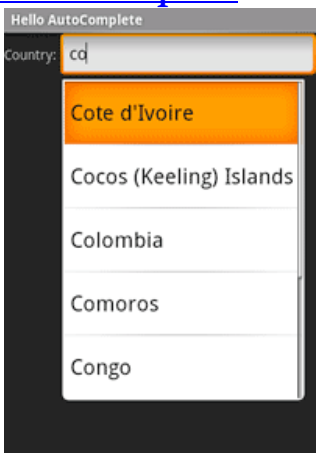
Form Stuff



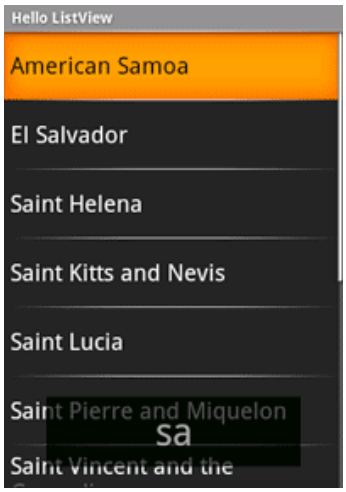
Spinner



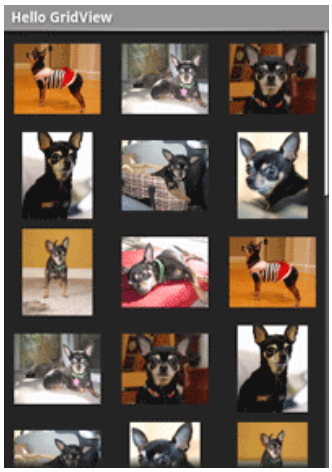
AutoComplete



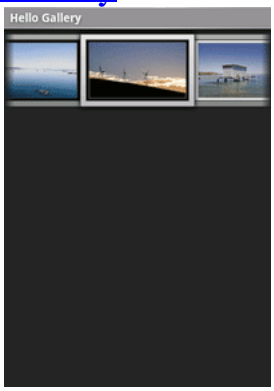
ListView



GridView

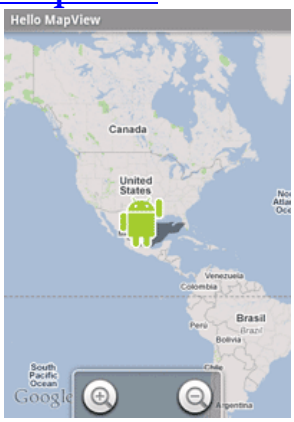


Gallery

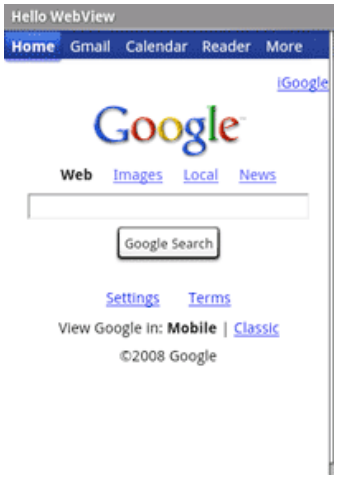


TabWidget

MapView



WebView



Hello, LinearLayout

A LinearLayout is a GroupView that will lay child View elements vertically or horizontally.

1. Start a new project/Activity called HelloLinearLayout.
2. Open the layout file. Make it like so:

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

    <LinearLayout
         android:orientation="horizontal"
         android:layout_width="fill_parent"
         android:layout_height="fill_parent"
         android:layout_weight="1">

         <TextView
             android:text="red"
             android:gravity="center_horizontal"
             android:background="#aa0000"
             android:layout_width="wrap_content"
             android:layout_height="fill_parent"
             android:layout_weight="1"/>

         <TextView
             android:text="green"
             android:gravity="center_horizontal"
             android:background="#00aa00"
             android:layout_width="wrap_content"
             android:layout_height="fill_parent"
             android:layout_weight="1"/>

         <TextView
             android:text="blue"
             android:gravity="center_horizontal"
             android:background="#0000aa"
             android:layout_width="wrap_content"
             android:layout_height="fill_parent"
             android:layout_weight="1"/>

         <TextView
             android:text="yellow"
             android:gravity="center_horizontal"
             android:background="#aaaa00"
             android:layout_width="wrap_content"
             android:layout_height="fill_parent"
             android:layout_weight="1"/>

     </LinearLayout>

```

```
49.      <LinearLayout
50.          android:orientation="vertical"
51.          android:layout_width="fill_parent"
52.          android:layout_height="fill_parent"
53.          android:layout_weight="1">
54.
55.          <TextView
56.              android:text="row one"
57.              android:textSize="15pt"
58.              android:layout_width="fill_parent"
59.              android:layout_height="wrap_content"
60.              android:layout_weight="1"/>
61.
62.          <TextView
63.              android:text="row two"
64.              android:textSize="15pt"
65.              android:layout_width="fill_parent"
66.              android:layout_height="wrap_content"
67.              android:layout_weight="1"/>
68.
69.          <TextView
70.              android:text="row three"
71.              android:textSize="15pt"
72.              android:layout_width="fill_parent"
73.              android:layout_height="wrap_content"
74.              android:layout_weight="1"/>
75.
76.          <TextView
77.              android:text="row four"
78.              android:textSize="15pt"
79.              android:layout_width="fill_parent"
80.              android:layout_height="wrap_content"
81.              android:layout_weight="1"/>
82.
83.      </LinearLayout>
84.
85.  </LinearLayout>
```

Carefully inspect the XML. You'll notice how this layout works a lot like an HTML layout. There is one parent LinearLayout that is defined to lay its child elements vertically. The first child is another LinearLayout that uses a horizontal layout and the second uses a vertical layout. Each LinearLayout contains several TextView elements.

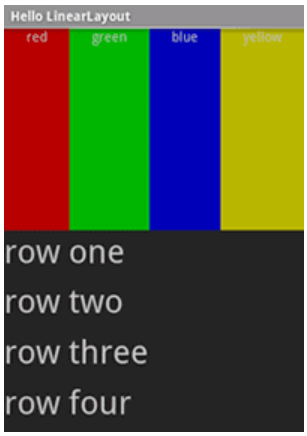
86. Now open the HelloLinearLayout Activity and be sure it loads this layout in the `onCreate()` method:

```
87.  public void onCreate(Bundle savedInstanceState) {
88.      super.onCreate(savedInstanceState);
89.      setContentView(R.layout.main);
90.  }
```

`R.layout.main` refers to the `main.xml` layout file.

91. Run it.

You should see the following:



Notice how the various XML attributes define the View's behavior. Pay attention to the effect of the `layout_weight`. Try experimenting with different values to see how the screen real estate is distributed based on the weight of each element.

Hello, RelativeLayout

A RelativeLayout is a ViewGroup that allows you to layout child elements in positions relative to the parent or siblings elements.

1. Start a new project/Activity called HelloRelativeLayout.
2. Open the layout file. Make it like so:

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

    <TextView
        android:id="@+id/label"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="Type here:"/>

     <EditText
         android:id="@+id/entry"
         android:layout_width="fill_parent"
         android:layout_height="wrap_content"
         android:background="@android:drawable/editbox_background"
         android:layout_below="@id/label"/>

     <Button
         android:id="@+id/ok"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_below="@id/entry"
         android:layout_alignParentRight="true"
         android:layout_marginLeft="10dip"
         android:text="OK" />

     <Button
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_toLeftOf="@id/ok"
         android:layout_alignTop="@id/ok"
         android:text="Cancel" />

 </RelativeLayout>
```

Pay attention to each of the additional `layout_*` attributes (besides the usual width and height, which are required for all elements). When using relative layout, we use attributes like `layout_below` and `layout_toLeftOf` to describe how we'd like to position each View. Naturally, these are different relative positions, and the value of the attribute is the id of the element we want the position relative to.

38. Make sure your Activity loads this layout in the `onCreate()` method:

```
39.  public void onCreate(Bundle savedInstanceState) {
40.      super.onCreate(savedInstanceState);
41.      setContentView(R.layout.main);
42.  }
```

`R.layout.main` refers to the `main.xml` layout file.

43. Run it.

You should see the following:

Hello, TableLayout

A TableLayout is a ViewGroup that will lay child View elements into rows and columns.

1. Start a new project/Activity called HelloTableLayout.
2. Open the layout file. Make it like so:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">

    <TableRow>
        <TextView
            android:layout_column="1"
            android:text="Open..."
            android:padding="3dip" />
        <TextView
            android:text="Ctrl-O"
            android:gravity="right"
            android:padding="3dip" />
     </TableRow>

     <TableRow>
        <TextView
            android:layout_column="1"
            android:text="Save..."
            android:padding="3dip" />
        <TextView
            android:text="Ctrl-S"
            android:gravity="right"
            android:padding="3dip" />
     </TableRow>

     <TableRow>
        <TextView
            android:layout_column="1"
            android:text="Save As..."
            android:padding="3dip" />
        <TextView
            android:text="Ctrl-Shift-S"
            android:gravity="right"
            android:padding="3dip" />
     </TableRow>

     <View
        android:layout_height="2dip"
        android:background="#FF909090" />

     <TableRow>
        <TextView
            android:text="X"
            android:padding="3dip" />
```

```
50.          <TextView
51.              android:text="Import..."
52.              android:padding="3dip" />
53.      </TableRow>
54.
55.      <TableRow>
56.          <TextView
57.              android:text="X"
58.              android:padding="3dip" />
59.          <TextView
60.              android:text="Export..."
61.              android:padding="3dip" />
62.          <TextView
63.              android:text="Ctrl-E"
64.              android:gravity="right"
65.              android:padding="3dip" />
66.      </TableRow>
67.
68.      <View
69.          android:layout_height="2dip"
70.          android:background="#FF909090" />
71.
72.      <TableRow>
73.          <TextView
74.              android:layout_column="1"
75.              android:text="Quit"
76.              android:padding="3dip" />
77.      </TableRow>
78.  </TableLayout>
```

Notice how this resembles the structure of an HTML table. `TableLayout` is like the `table` element; `TableRow` is like a `tr` element; but for our cells like the html `td` element, we can use any kind of View. Here, we use `TextView` for the cells.

79. Make sure your Activity loads this layout in the `onCreate()` method:

```
80. public void onCreate(Bundle savedInstanceState) {
81.     super.onCreate(savedInstanceState);
82.     setContentView(R.layout.main);
83. }
```

`R.layout.main` refers to the `main.xml` layout file.

84. Run it.

You should see the following:

Hello, Form Stuff

This page introduces a variety of widgets, like image buttons, text fields, checkboxes and radio buttons.

1. Start a new project/Activity called HelloFormStuff.
2. Your layout file should have a basic LinearLayout:

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

</LinearLayout>
```

   For each widget you want to add, just put the respective View inside here.

Tip: As you add new Android code, press Ctrl(or Cmd) + Shift + O to import all needed packages.

ImageButton

A button with a custom image on it. We'll make it display a message when pressed.

1. Drag the Android image on the right (or your own image) into the res/drawable/ directory of your project. We'll use this for the button.
2. Open the layout file and, inside the LinearLayout, add the ImageButton element:

```
<ImageButton
    android:id="@+id/android_button"
    android:layout_width="100dip"
    android:layout_height="wrap_content"
    android:src="@drawable/android" />
```

   The source of the button is from the res/drawable/ directory, where we've placed the android.png.

Tip: You can also reference some of the many built-in images from the Android R.drawable resources, like `ic_media_play`, for a "play" button image. To do so, change the source attribute to `android:src="@android:drawable/ic_media_play"`.

8. To make the button to actually do something, add the following code at the end of the `onCreate()` method:

```
final ImageButton button = (ImageButton)
findViewById(R.id.android_button);
 button.setOnClickListener(new OnClickListener() {
     public void onClick(View v) {
         // Perform action on clicks
         Toast.makeText(HelloFormStuff.this, "Beep Bop",
Toast.LENGTH_SHORT).show();
     }
 });
```

This captures our ImageButton from the layout, then adds an on-click listener to it. The View.OnClickListener must define the `onClick()` method, which defines the action to be made when the button is clicked. Here, we show a Toast message when clicked.

16. Run it.

EditText

A text field for user input. We'll make it display the text entered so far when the "Enter" key is pressed.

1. Open the layout file and, inside the LinearLayout, add the EditText element:

```
<EditText
    android:id="@+id/edittext"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
```

6. To do something with the text that the user enters, add the following code to the end of the `onCreate()` method:

```
final EditText edittext = (EditText) findViewById(R.id.edittext);
edittext.setOnKeyListener(new OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent event) {
        if ((event.getAction() == KeyEvent.ACTION_DOWN) && (keyCode ==
KeyEvent.KEYCODE_ENTER)) {
          // Perform action on key press
          Toast.makeText(HelloFormStuff.this, edittext.getText(),
Toast.LENGTH_SHORT).show();
          return true;
        }
        return false;
    }
 });
```

This captures our EditText element from the layout, then adds an on-key listener to it. The View.OnKeyListener must define the `onKey()` method, which defines the action to

be made when a key is pressed. In this case, we want to listen for the Enter key (when pressed down), then pop up a Toast message with the text from the EditText field. Be sure to return *true* after the event is handled, so that the event doesn't bubble-up and get handled by the View (which would result in a carriage return in the text field).

18. Run it.

CheckBox

A checkbox for selecting items. We'll make it display the the current state when pressed.

1. Open the layout file and, inside the LinearLayout, add the CheckBox element:

```
<CheckBox android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="check it out" />
```

6. To do something when the state is changed, add the following code to the end of the `onCreate()` method:

```
final CheckBox checkbox = (CheckBox) findViewById(R.id.checkbox);
checkbox.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        // Perform action on clicks
        if (checkbox.isChecked()) {
            Toast.makeText(HelloFormStuff.this, "Selected",
Toast.LENGTH_SHORT).show();
        } else {
            Toast.makeText(HelloFormStuff.this, "Not selected",
Toast.LENGTH_SHORT).show();
        }
    }
 });
```

This captures our CheckBox element from the layout, then adds an on-click listener to it. The View.OnClickListener must define the `onClick()` method, which defines the action to be made when the checkbox is clicked. Here, we query the current state of the checkbox, then pop up a Toast message that displays the current state. Notice that the CheckBox handles its own state change between checked and un-checked, so we just ask which it currently is.

18. Run it.

Tip: If you find that you need to change the state in another way (such as when loading a saved CheckBoxPreference), use `setChecked(true)` or `toggle()`.

RadioButton

Two mutually-exclusive radio buttons—enabling one disables the other. When each is pressed, we'll pop up a message.

1. Open the layout file and, inside the LinearLayout, add two RadioButtons, inside a RadioGroup:

```
<RadioGroup
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">

  <RadioButton android:id="@+id/radio_red"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
       android:text="Red" />

   <RadioButton android:id="@+id/radio_blue"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Blue" />

</RadioGroup>
```

18. To do something when each is selected, we'll need an OnClickListener. Unlike the other listeners we've created, instead of creating this one as an anonymous inner class, we'll create it as a new object. This way, we can re-use the OnClickLIstener for both RadioButtons. So, add the following code in the HelloFormStuff Activity (*outside* the `onCreate()` method):

```
OnClickListener radio_listener = new OnClickListener() {
    public void onClick(View v) {
        // Perform action on clicks
        RadioButton rb = (RadioButton) v;
        Toast.makeText(HelloFormStuff.this, rb.getText(),
Toast.LENGTH_SHORT).show();
    }
};
```

Our `onClick()` method will be handed the View clicked, so the first thing to do is cast it into a RadioButton. Then we pop up a Toast message that displays the selection.

26. Now, at the bottom of the `onCreate()` method, add the following:

```
   final RadioButton radio_red = (RadioButton)
findViewById(R.id.radio_red);
   final RadioButton radio_blue = (RadioButton)
findViewById(R.id.radio_blue);
   radio_red.setOnClickListener(radio_listener);
   radio_blue.setOnClickListener(radio_listener);
```

This captures each of the RadioButtons from our layout and adds the newly-created OnClickListener to each.

31. Run it.

Tip: If you find that you need to change the state of a RadioButton in another way (such as when loading a saved CheckBoxPreference), use `setChecked(true)` or `toggle()`.

ToggleButton

A button used specifically for toggling something on and off.

1. Open the layout file and, inside the LinearLayout, add the ToggleButton element:

```
<ToggleButton android:id="@+id/togglebutton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

5. To do something when the state is changed, add the following code to the end of the `onCreate()` method:

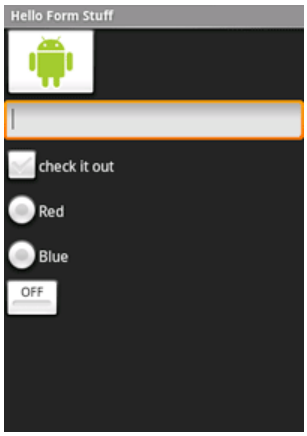
```
final ToggleButton togglebutton = (ToggleButton)
findViewById(R.id.togglebutton);
togglebutton.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        // Perform action on clicks
          if (togglebutton.isChecked()) {
              Toast.makeText(HelloFormStuff.this, "ON",
Toast.LENGTH_SHORT).show();
          } else {
              Toast.makeText(HelloFormStuff.this, "OFF",
Toast.LENGTH_SHORT).show();
          }
      }
 });
```

   This captures our ToggleButton element from the layout, then adds an on-click listener to it. The View.OnClickListener must define the `onClick()` method, which defines the action to be made when the button is clicked. Here, we query the current state of the ToggleButton, then pop up a Toast message that displays the current state. Notice that the ToggleButton handles its own state change between checked and un-checked, so we just ask which it is.

17. Run it.

Tip: By default, the text on the button is "ON" and "OFF", but you can change each of these with `setTextOn(`*`CharSequence`*`)` and `setTextOff(`*`CharSequence`*`)`. And, if you find that you need to change the state in another way (such as when loading a saved CheckBoxPreference), use `setChecked(true)` or `toggle()`.

If you've added all the form items above, your application should look something like this:



Button

extends TextView

java.lang.Object
↳ android.view.View
↳ android.widget.TextView
↳ android.widget.Button

Class Overview

`Button` represents a push-button widget. Push-buttons can be pressed, or clicked, by the user to perform an action. A typical use of a push-button in an activity would be the following:

```
public class MyActivity extends Activity {
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        setContentView(R.layout.content_layout_id);

        final Button button = (Button) findViewById(R.id.button_id);
        button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                // Perform action on click
            }
        });
    }
}
```

ToggleButton

Class Overview

Displays checked/unchecked states as a button with a "light" indicator and by default accompanied with the text "ON" or "OFF".

Creating Menus

Key classes

1. Menu
2. ContextMenu
3. SubMenu

In this document

Menus are an important part of any application. They provide familiar interfaces that reveal application functions and settings. Android offers an easy programming interface for developers to provide standardized application menus for various situations.

Android offers three fundamental types of application menus:

Options Menu

This is the primary set of menu items for an Activity. It is revealed by pressing the device MENU key. Within the Options Menu are two groups of menu items:

Icon Menu

This is the collection of items initially visible at the bottom of the screen at the press of the MENU key. It supports a maximum of six menu items. These are the only menu items that support icons and the only menu items that *do not* support checkboxes or radio buttons.

Expanded Menu

This is a vertical list of items exposed by the "More" menu item from the Icon Menu. It exists only when the Icon Menu becomes over-loaded and is comprised of the sixth Option Menu item and the rest.

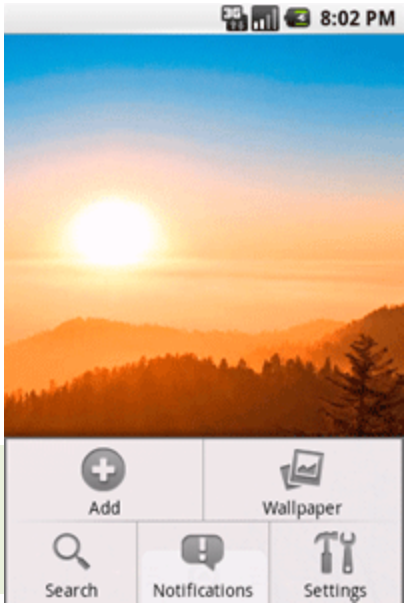
Context Menu

This is a floating list of menu items that may appear when you perform a long-press on a View (such as a list item).

Submenu

This is a floating list of menu items that is revealed by an item in the Options Menu or a Context Menu. A Submenu item cannot support nested Submenus.

Options Menu



The Options Menu is opened by pressing the device MENU key. When opened, the Icon Menu is displayed, which holds the first six menu items. If more than six items are added to the Options Menu, then those that can't fit in the Icon Menu are revealed in the Expanded Menu, via the "More" menu item. The Expanded Menu is automatically added when there are more than six items.

The Options Menu is where you should include basic application functions and any necessary navigation items (e.g., to a home screen or application settings). You can also add Submenus for organizing topics and including extra menu functionality.

When this menu is opened for the first time, the Android system will call the Activity `onCreateOptionsMenu()` callback method. Override this method in your Activity and populate the Menu object given to you.

You can populate the menu by inflating a menu resource that was defined in XML, or by calling `add()` for each item you'd like in the menu. This method adds a MenuItem, and returns the newly created object to you. You can use the returned MenuItem to set additional properties like an icon, a keyboard shortcut, an intent, and other settings for the item.

There are multiple `add()` methods. Usually, you'll want to use one that accepts an *itemId* argument. This is a unique integer that allows you to identify the item during a callback.

When a menu item is selected from the Options Menu, you will receive a callback to the `onOptionsItemSelected()` method of your Activity. This callback passes you the `MenuItem` that has been selected. You can identify the item by requesting the *itemId*, with `getItemId()`, which returns the integer that was assigned with the `add()` method. Once you identify the menu item, you can take the appropriate action.

Here's an example of this procedure, inside an Activity, wherein we create an Options Menu and handle item selections:

```
/* Creates the menu items */
public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(0, MENU_NEW_GAME, 0, "New Game");
    menu.add(0, MENU_QUIT, 0, "Quit");
    return true;
}

/* Handles item selections */
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case MENU_NEW_GAME:
        newGame();
        return true;
    case MENU_QUIT:
        quit();
        return true;
    }
    return false;
}
```

The `add()` method used in this sample takes four arguments: *groupId*, *itemId*, *order*, and *title*. The *groupId* allows you to associate this menu item with a group of other items (more about Menu groups, below) — in this example, we ignore it. *itemId* is a unique integer that we give the MenuItem so that can identify it in the next callback. *order* allows us to define the display order of the item — by default, they are displayed by the order in which we add them. *title* is, of course, the name that goes on the menu item (this can also be a string resource, and we recommend you do it that way for easier localization).

Tip: If you have several menu items that can be grouped together with a title, consider organizing them into a Submenu.

Adding icons

Icons can also be added to items that appears in the Icon Menu with `setIcon()`. For example:

```
menu.add(0, MENU_QUIT, 0, "Quit")
    .setIcon(R.drawable.menu_quit_icon);
```

Modifying the menu

If you want to sometimes re-write the Options Menu as it is opened, override the `onPrepareOptionsMenu()` method, which is called each time the menu is opened. This will pass you the Menu object, just like the `onCreateOptionsMenu()` callback. This is useful if you'd like to add or remove menu options depending on the current state of an application or game.

Note: When changing items in the menu, it's bad practice to do so based on the currently selected item. Keep in mind that, when in touch mode, there will not be a selected (or focused) item. Instead, you should use a Context Menu for such behaviors, when you want to provide functionality based on a particular item in the UI.

Context Menu

The Android context menu is similar, in concept, to the menu revealed with a "right-click" on a PC. When a view is registered to a context menu, performing a "long-press" (press and hold for about two seconds) on the object will reveal a floating menu that provides functions relating to that item. Context menus can be registered to any View object, however, they are most often used for items in a ListView, which helpfully indicates the presence of the context menu by transforming the background color of the ListView item when pressed. (The items in the phone's contact list offer an example of this feature.)

Note: Context menu items do not support icons or shortcut keys.

To create a context menu, you must override the Activity's context menu callback methods: `onCreateContextMenu()` and `onContextItemSelected()`. Inside the `onCreateContextMenu()` callback method, you can add menu items using one of the `add()` methods, or by inflating a menu resource that was defined in XML. Then, register a ContextMenu for the View, with `registerForContextMenu()`.

For example, here is some code that can be used with the Notepad application to add a context menu for each note in the list:

```
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenuInfo menuInfo) {
  super.onCreateContextMenu(menu, v, menuInfo);
  menu.add(0, EDIT_ID, 0, "Edit");
  menu.add(0, DELETE_ID, 0,  "Delete");
}

public boolean onContextItemSelected(MenuItem item) {
  AdapterContextMenuInfo info = (AdapterContextMenuInfo) item.getMenuInfo();
  switch (item.getItemId()) {
  case EDIT_ID:
    editNote(info.id);
    return true;
  case DELETE_ID:
    deleteNote(info.id);
    return true;
  default:
    return super.onContextItemSelected(item);
  }
}
```

In `onCreateContextMenu()`, we are given not only the ContextMenu to which we will add MenuItems, but also the View that was selected and a ContextMenuInfo object, which provides additional information about the object that was selected. In this example, nothing special is done in `onCreateContextMenu()` — just a couple items are added as usual. In the `onContextItemSelected()` callback, we request the AdapterContextMenuInfo from the `MenuItem`, which provides information about the currently selected item. All we need from this

is the list ID for the selected item, so whether editing a note or deleting it, we find the ID with the `AdapterContextMenuInfo.info` field of the object. This ID is passed to the `editNote()` and `deleteNote()` methods to perfrom the respective action.

Now, to register this context menu for all the items in a ListView, we pass the entire `ListView` to the `registerForContextMenu(View)` method:

```
registerForContextMenu(getListView());
```

Remember, you can pass any View object to register a context menu. Here, `getListView()` returns the ListView object used in the Notepad application's ListActivity. As such, each item in the list is registered to this context menu.

Submenus

A sub menu can be added within any menu, except another sub menu. These are very useful when your application has a lot of functions that may be organized in topics, like the items in a PC application's menu bar (File, Edit, View, etc.).

A sub menu is created by adding it to an existing Menu with `addSubMenu()`. This returns a SubMenu object (an extension of Menu). You can then add additional items to this menu, with the normal routine, using the `add()` methods. For example:

```
public boolean onCreateOptionsMenu(Menu menu) {
  boolean result = super.onCreateOptionsMenu(menu);

  SubMenu fileMenu = menu.addSubMenu("File");
  SubMenu editMenu = menu.addSubMenu("Edit");
  fileMenu.add("new");
  fileMenu.add("open");
  fileMenu.add("save");
  editMenu.add("undo");
  editMenu.add("redo");

  return result;
}
```

Callbacks for items selected in a sub menu are made to the parent menu's callback method. For the example above, selections in the sub menu will be handled by the `onOptionsItemSelected()` callback.

You can also add Submenus when you define the parent menu in XML.

Define Menus in XML

Just like Android UI layouts, you can define application menus in XML, then inflate them in your menu's `onCreate...()` callback method. This makes your application code cleaner and separates more interface design into XML, which is easier to visualize.

To start, create a new folder in your project `res/` directory called `menu`. This is where you should keep all XML files that define your application menus.

In a menu XML layout, there are three valid elements: `<menu>`, `<group>` and `<item>`. The `item` and `group` elements must be children of a `menu`, but `item` elements may also be the children of a `group`, and another `menu` element may be the child of an `item` (to create a Submenu). Of course, the root node of any file must be a `menu` element.

As an example, we'll define the same menu created in the Options Menu section, above. We start with an XML file named `options_menu.xml` inside the `res/menu/` folder:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/new_game"
          android:title="New Game" />
    <item android:id="@+id/quit"
          android:title="Quit" />
</menu>
```

Then, in the `onCreateOptionsMenu()` method, we inflate this resource using `MenuInflater.inflate()`:

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.options_menu, menu);
    return true;
}
```

The `getMenuInflater()` method returns the MenuInflater for our activity's context. We then call `inflate()`, passing it a pointer to our menu resource and the Menu object given by the callback.

While this small sample may seem like more effort, compared to creating the menu items in the `onCreateOptionsMenu()` method, this will save a lot of trouble when dealing with more items and it keeps your application code clean.

You can define menu groups by wrapping `item` elements in a `group` element, and create Submenus by nesting another `menu` inside an `item`. Each element also supports all the necessary attributes to control features like shortcut keys, checkboxes, icons, and more. To learn about these attributes and more about the XML syntax, see the Menus topic in the Available Resource Types document.

Menu Features

Here are some other features that can be applied to most menu items.

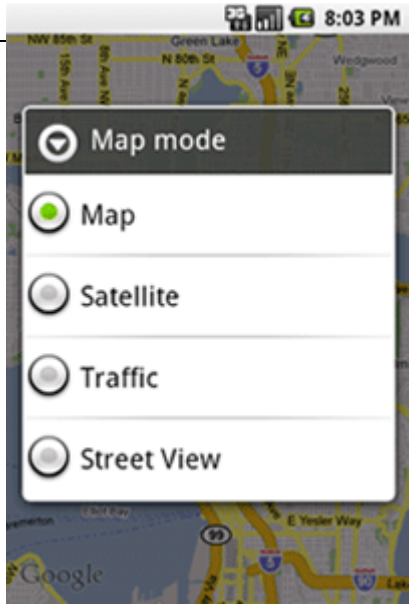
Menu groups

When adding new items to a menu, you can optionally include each item in a group. A menu group is a collection of menu items that can share certain traits, like whether they are visible, enabled, or checkable.

A group is defined by an integer (or a resource id, in XML). A menu item is added to the group when it is added to the menu, using one of the `add()` methods that accepts a *groupId* as an argument, such as `add(int, int, int, int)`.

You can show or hide the entire group with `setGroupVisible()`; enable or disable the group with `setGroupEnabled()`; and set whether the items can be checkable with `setGroupCheckable()`.

Checkable menu items



Any menu item can be used as an interface for turning options on and off. This can be indicated with a checkbox for stand-alone options, or radio buttons for groups of mutually exlusive options (see the screenshot, to the right).

Note: Menu items in the Icon Menu cannot display a checkbox or radio button. If you choose to make items in the Icon Menu checkable, then you must personally indicate the state by swapping the icon and/or text each time the state changes between on and off.

To make a single item checkable, use the `setCheckable()` method, like so:

```
menu.add(0, VIBRATE_SETTING_ID, 0, "Vibrate")
    .setCheckable(true);
```

This will display a checkbox with the menu item (unless it's in the Icon Menu). When the item is selected, the `onOptionsItemSelected()` callback is called as usual. It is here that you must set the state of the checkbox. You can query the current state of the item with `isChecked()` and set the checked state with `setChecked()`. Here's what this looks like inside the `onOptionsItemSelected()` callback:

```
switch (item.getItemId()) {
case VIBRATE_SETTING_ID:
  if (item.isChecked()) item.setChecked(false);
  else item.setChecked(true);
  return true;
...
}
```

To make a group of mutually exclusive radio button items, simply assign the same group ID to each menu item and call `setGroupCheckable()`. In this case, you don't need to call `setCheckable()` on each menu items, because the group as a whole is set checkable. Here's an example of two mutually exclusive options in a Submenu:

```
SubMenu subMenu = menu.addSubMenu("Color");
subMenu.add(COLOR_MENU_GROUP, COLOR_RED_ID, 0, "Red");
subMenu.add(COLOR_MENU_GROUP, COLOR_BLUE_ID, 0, "Blue");
subMenu.setGroupCheckable(COLOR_MENU_GROUP, true, true);
```

In the `setGroupCheckable()` method, the first argument is the group ID that we want to set checkable. The second argument is whether we want the group items to be checkable. The last one is whether we want each item to be exclusively checkable (if we set this *false*, then all the items will be checkboxes instead of radio buttons). When the group is set to be exclusive (radio buttons), each time a new item is selected, all other are automatically de-selected.

Note: Checkable menu items are intended to be used only on a per-session basis and not saved to the device (e.g., the *Map mode* setting in the Maps application is not saved — screenshot above). If there are application settings that you would like to save for the user, then you should store the data using Preferences, and manage them with a PreferenceActivity.

Shortcut keys

Quick access shortcut keys using letters and/or numbers can be added to menu items with `setAlphabeticShortcut(char)` (to set char shortcut), `setNumericShortcut(int)` (to set numeric shortcut), or `setShortcut(char,int)` (to set both). Case is *not* sensitive. For example:

```
menu.add(0, MENU_QUIT, 0, "Quit")
    .setAlphabeticShortcut('q');
```

Now, when the menu is open (or while holding the MENU key), pressing the "q" key will select this item.

This shortcut key will be displayed as a tip in the menu item, below the menu item name (except for items in the Icon Menu).

Note: Shortcuts cannot be added to items in a Context Menu.

Menu item intents

If you've read the Application Fundamentals, then you're at least a little familiar with Android Intents. These allow applications to bind with each other, share information, and perform user tasks cooperatively. Just like your application might fire an Intent to launch a web browser, an email client, or another Activity in your application, you can perform such actions from within a menu. There are two ways to do this: define an Intent and assign it to a single menu item, or define an Intent and allow Android to search the device for activities and dynamically add a menu item for each one that meets the Intent criteria.

For more information on creating Intents and providing your application's services to other applications, read the Intents and Intent Filters document.

Set an intent for a single menu item

If you want to offer a specific menu item that launches a new Activity, then you can specifically define an Intent for the menu item with the `setIntent()` method.

For example, inside the `onCreateOptionsMenu()` method, you can define a new menu item with an Intent like this:

```
MenuItem menuItem = menu.add(0, PHOTO_PICKER_ID, 0, "Select Photo");
menuItem.setIntent(new Intent(this, PhotoPicker.class));
```

Android will automatically launch the Activity when the item is selected.

Note: This will not return a result to your Activity. If you wish to be returned a result, then do not use `setIntent()`. Instead, handle the selection as usual in the `onOptionsMenuItemSelected()` or `onContextMenuItemSelected()` callback and call `startActivityForResult()`.

Dynamically add intents

If there are potentially multiple activities that are relevant to your current Activity or selected item, then the application can dynamically add menu items that execute other services.

During menu creation, define an Intent with the category *Intent.ALTERNATIVE_CATEGORY* and/or *Intent.SELECTED_ALTERNATIVE*, the MIME type currently selected (if any), and any other requirements, the same way as you would satisfy an intent filter to open a new Activity. Then call `addIntentOptions()` to have Android search for any services meeting those requirements and add them to the menu for you. If there are no applications installed that satisfy the Intent, then no additional menu items are added.

Note: *SELECTED_ALTERNATIVE* is used to handle the currently selected element on the screen. So, it should only be used when creating a Menu in `onCreateContextMenu()` or `onPrepareOptionsMenu()`, which is called every time the Options Menu is opened.

Here's an example demonstrating how an application would search for additional services to display on its menu.

```
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);

    // Create an Intent that describes the requirements to fulfill, to be
included
    // in our menu. The offering app must include a category value of
Intent.CATEGORY_ALTERNATIVE.
    Intent intent = new Intent(null, getIntent().getData());
    intent.addCategory(Intent.CATEGORY_ALTERNATIVE);

    // Search for, and populate the menu with, acceptable offering
applications.
    menu.addIntentOptions(
         thisClass.INTENT_OPTIONS,  // Menu group
         0,      // Unique item ID (none)
         0,      // Order for the items (none)
         this.getComponentName(),   // The current Activity name
         null,   // Specific items to place first (none)
         intent, // Intent created above that describes our requirements
         0,      // Additional flags to control items (none)
         null);  // Array of MenuItems that corrolate to specific items
(none)

    return true;
}
```

For each Activity found that provides an Intent Filter matching the Intent defined, a menu item will be added, using the *android:label* value of the intent filter as the text for the menu item. The `addIntentOptions()` method will also return the number of menu items added.

Also be aware that, when `addIntentOptions()` is called, it will override any and all menu items in the menu group specified in the first argument.

If you wish to offer the services of your Activity to other application menus, then you only need to define an intent filter as usual. Just be sure to include the *ALTERNATIVE* and/or *SELECTED_ALTERNATIVE* values in the *name* attribute of a `<category>` element in the intent filter. For example:

```
<intent-filter label="Resize Image">
    ...
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
    ...
</intent-filter>
```

Binding to Data with AdapterView

In this document

See also

1. Hello Spinner tutorial
2. Hello ListView tutorial
3. Hello GridView tutorial

The AdapterView is a ViewGroup subclass whose child Views are determined by an Adapter that binds to data of some type. AdapterView is useful whenever you need to display stored data (as opposed to resource strings or drawables) in your layout.

Gallery, ListView, and Spinner are examples of AdapterView subclasses that you can use to bind to a specific type of data and display it in a certain way.

AdapterView objects have two main responsibilities:

- Filling the layout with data
- Handling user selections

Filling the Layout with Data

Inserting data into the layout is typically done by binding the AdapterView class to an Adapter, which retireves data from an external source (perhaps a list that the code supplies or query results from the device's database).

The following code sample does the following:

1. Creates a Spinner with an existing View and binds it to a new ArrayAdapter that reads an array of colors from the local resources.
2. Creates another Spinner object from a View and binds it to a new SimpleCursorAdapter that will read people's names from the device contacts (see Contacts.People).

```
// Get a Spinner and bind it to an ArrayAdapter that
// references a String array.
Spinner s1 = (Spinner) findViewById(R.id.spinner1);
ArrayAdapter adapter = ArrayAdapter.createFromResource(
    this, R.array.colors, android.R.layout.simple_spinner_item);
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
s1.setAdapter(adapter);

// Load a Spinner and bind it to a data query.
```

```
private static String[] PROJECTION = new String[] {
        People._ID, People.NAME
    };

Spinner s2 = (Spinner) findViewById(R.id.spinner2);
Cursor cur = managedQuery(People.CONTENT_URI, PROJECTION, null, null);

SimpleCursorAdapter adapter2 = new SimpleCursorAdapter(this,
    android.R.layout.simple_spinner_item, // Use a template
                                          // that displays a
                                          // text view
    cur, // Give the cursor to the list adatper
    new String[] {People.NAME}, // Map the NAME column in the
                                // people database to...
    new int[] {android.R.id.text1}); // The "text1" view defined in
                                     // the XML template

adapter2.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
s2.setAdapter(adapter2);
```

Note that it is necessary to have the People._ID column in projection used with CursorAdapter or else you will get an exception.

If, during the course of your application's life, you change the underlying data that is read by your Adapter, you should call notifyDataSetChanged(). This will notify the attached View that the data has been changed and it should refresh itself.

Handling User Selections

You handle the user's selecction by setting the class's AdapterView.OnItemClickListener member to a listener and catching the selection changes.

```
// Create a message handling object as an anonymous class.
private OnItemClickListener mMessageClickedHandler = new OnItemClickListener() {
    public void onItemClick(AdapterView parent, View v, int position, long id)
    {
        // Display a messagebox.
        Toast.makeText(mContext,"You've got an event",Toast.LENGTH_SHORT).show();
    }
};

// Now hook into our object and set its onItemClickListener member
// to our class handler object.
mHistoryView = (ListView)findViewById(R.id.history);
mHistoryView.setOnItemClickListener(mMessageClickedHandler);
```

For more discussion on how to create different AdapterViews, read the following tutorials: Hello Spinner, Hello ListView, and Hello GridView.

Handling UI Events

In this document

See also

1. Hello Form Stuff tutorial

On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific View object that the user interacts with. The View class provides the means to do so.

Within the various View classes that you'll use to compose your layout, you may notice several public callback methods that look useful for UI events. These methods are called by the Android framework when the respective action occurs on that object. For instance, when a View (such as a Button) is touched, the `onTouchEvent()` method is called on that object. However, in order to intercept this, you must extend the class and override the method. Obviously, extending every View object you want to use (just to handle an event) would be obsurd. This is why the View class also contains a collection of nested interfaces with callbacks that you can much more easily define. These interfaces, called event listeners, are your ticket to capturing the user interaction with your UI.

While you will more commonly use the event listeners to listen for user interaction, there may come a time when you do want to extend a View class, in order to build a custom component. Perhaps you want to extend the Button class to make something more fancy. In this case, you'll be able to define the default event behaviors for your class using the class event handlers.

Event Listeners

An event listener is an interface in the View class that contains a single callback method. These methods will be called by the Android framework when the View to which the listener has been registered is triggered by user interaction with the item in the UI.

Included in the event listener interfaces are the following callback methods:

`onClick()`

> From View.OnClickListener. This is called when the user either touches the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses the suitable "enter" key or presses down on the trackball.

`onLongClick()`

> From View.OnLongClickListener. This is called when the user either touches and holds the item (when in touch mode), or focuses upon the item with the navigation-keys or trackball and presses and holds the suitable "enter" key or presses and holds down on the trackball (for one second).

`onFocusChange()`

> From View.OnFocusChangeListener. This is called when the user navigates onto or away from the item, using the navigation-keys or trackball.

`onKey()`

> From View.OnKeyListener. This is called when the user is focused on the item and presses or releases a key on the device.

`onTouch()`

> From View.OnTouchListener. This is called when the user performs an action qualified as a touch event, including a press, a release, or any movement gesture on the screen (within the bounds of the item).

`onCreateContextMenu()`

> From View.OnCreateContextMenuListener. This is called when a Context Menu is being built (as the result of a sustained "long click"). See the discussion on context menus in Creating Menus for more information.

These methods are the sole inhabitants of their respective interface. To define one of these methods and handle your events, implement the nested interface in your Activity or define it as an anonymous class. Then, pass an instance of your implementation to the respective `View.set...Listener()` method. (E.g., call `setOnClickListener()` and pass it your implementation of the OnClickListener.)

The example below shows how to register an on-click listener for a Button.

```
// Create an anonymous implementation of OnClickListener
private OnClickListener mCorkyListener = new OnClickListener() {
    public void onClick(View v) {
      // do something when the button is clicked
    }
```

```
};

protected void onCreate(Bundle savedValues) {
    ...
    // Capture our button from layout
    Button button = (Button)findViewById(R.id.corky);
    // Register the onClick listener with the implementation above
    button.setOnClickListener(mCorkyListener);
    ...
}
```

You may also find it more conventient to implement OnClickListener as a part of your Activity. This will avoid the extra class load and object allocation. For example:

```
public class ExampleActivity extends Activity implements OnClickListener {
    protected void onCreate(Bundle savedValues) {
        ...
        Button button = (Button)findViewById(R.id.corky);
        button.setOnClickListener(this);
    }

    // Implement the OnClickListener callback
    public void onClick(View v) {
      // do something when the button is clicked
    }
    ...
}
```

Notice that the `onClick()` callback in the above example has no return value, but some other event listener methods must return a boolean. The reason depends on the event. For the few that do, here's why:

- `onLongClick()` - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return *true* to indicate that you have handled the event and it should stop here; return *false* if you have not handled it and/or the event should continue to any other on-click listeners.
- `onKey()` - This returns a boolean to indicate whether you have consumed the event and it should not be carried further. That is, return *true* to indicate that you have handled the event and it should stop here; return *false* if you have not handled it and/or the event should continue to any other on-key listeners.
- `onTouch()` - This returns a boolean to indicate whether your listener consumes this event. The important thing is that this event can have multiple actions that follow each other. So, if you return *false* when the down action event is received, you indicate that you have not consumed the event and are also not interested in subsequent actions from this event. Thus, you will not be called for any other actions within the event, such as a fingure gesture, or the eventual up action event.

Remember that key events are always delivered to the View currently in focus. They are dispatched starting from the top of the View hierarchy, and then down, until they reach the appropriate destination. If your View (or a child of your View) currently has focus, then you can see the event travel through the `dispatchKeyEvent()` method. As an alternative to capturing

key events through your View, you can also receive all of the events inside your Activity with `onKeyDown()` and `onKeyUp()`.

Note: Android will call event handlers first and then the appropriate default handlers from the class definition second. As such, returning *true* from these event listeners will stop the propagation of the event to other event listeners and will also block the callback to the default event handler in the View. So be certain that you want to terminate the event when you return *true*.

Event Handlers

If you're building a custom component from View, then you'll be able to define several callback methods used as default event handlers. In the document on Building Custom Components, you'll learn see some of the common callbacks used for event handling, including:

- `onKeyDown(int, KeyEvent)` - Called when a new key event occurs.
- `onKeyUp(int, KeyEvent)` - Called when a key up event occurs.
- `onTrackballEvent(MotionEvent)` - Called when a trackball motion event occurs.
- `onTouchEvent(MotionEvent)` - Called when a touch screen motion event occurs.
- `onFocusChanged(boolean, int, Rect)` - Called when the view gains or loses focus.

There are some other methods that you should be awere of, which are not part of the View class, but can directly impact the way you're able to handle events. So, when managing more complex events inside a layout, consider these other methods:

- `Activity.dispatchTouchEvent(MotionEvent)` - This allows your Activity to intercept all touch events before they are dispatched to the window.
- `ViewGroup.onInterceptTouchEvent(MotionEvent)` - This allows a ViewGroup to watch events as they are dispatched to child Views.
- `ViewParent.requestDisallowInterceptTouchEvent(boolean)` - Call this upon a parent View to indicate that it should not intercept touch events with `onInterceptTouchEvent(MotionEvent)`.

Touch Mode

When a user is navigating a user interface with directional keys or a trackball, it is necessary to give focus to actionable items (like buttons) so the user can see what will accept input. If the device has touch capabilities, however, and the user begins interacting with the interface by touching it, then it is no longer necessary to highlight items, or give focus to a particular View. Thus, there is a mode for interaction named "touch mode."

For a touch-capable device, once the user touches the screen, the device will enter touch mode. From this point onward, only Views for which isFocusableInTouchMode() is true will be focusable, such as text editing widgets. Other Views that are touchable, like buttons, will not take focus when touched; they will simply fire their on-click listeners when pressed.

Any time a user hits a directional key or scrolls with a trackball, the device will exit touch mode, and find a view to take focus. Now, the user may resume interacting with the user interface without touching the screen.

The touch mode state is maintained throughout the entire system (all windows and activities). To query the current state, you can call isInTouchMode() to see whether the device is currently in touch mode.

Handling Focus

The framework will handle routine focus movement in response to user input. This includes changing the focus as Views are removed or hidden, or as new Views become available. Views indicate their willingness to take focus through the `isFocusable()` method. To change whether a View can take focus, call `setFocusable()`. When in touch mode, you may query whether a View allows focus with `isFocusableInTouchMode()`. You can change this with `setFocusableInTouchMode()`.

Focus movement is based on an algorithm which finds the nearest neighbor in a given direction. In rare cases, the default algorithm may not match the intended behavior of the developer. In these situations, you can provide explicit overrides with the following XML attributes in the layout file: *nextFocusDown*, *nextFocusLeft*, *nextFocusRight*, and *nextFocusUp*. Add one of these attributes to the View *from* which the focus is leaving. Define the value of the attribute to be the id of the View *to* which focus should be given. For example:

```
<LinearLayout
    android:orientation="vertical"
    ... >
  <Button android:id="@+id/top"
          android:nextFocusUp="@+id/bottom"
          ... />
  <Button android:id="@+id/bottom"
          android:nextFocusDown="@+id/top"
          ... />
</LinearLayout>
```

Ordinarily, in this vertical layout, navigating up from the first Button would not go anywhere, nor would navigating down from the second Button. Now that the top Button has defined the bottom one as the *nextFocusUp* (and vice versa), the navigation focus will cycle from top-to-bottom and bottom-to-top.

If you'd like to declare a View as focusable in your UI (when it is traditionally not), add the `android:focusable` XML attribute to the View, in your layout declaration. Set the value *true*. You can also declare a View as focusable while in Touch Mode with `android:focusableInTouchMode`.

To request a particular View to take focus, call `requestFocus()`.

To listen for focus events (be notified when a View receives or looses focus), use `onFocusChange()`, as discussed in the Event Listeners section, above.