Declaring Layout

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 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.

- The 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 layoutopt tool lets you quickly analyze your layouts and hierarchies for inefficiencies or other problems.

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

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,

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

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.

> and attributes, as well as the format of the XML file, see Layout Resources.

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 about lifecycles, in the Activities document).

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"/>
```

2. 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 figure 1, the parent view group defines layout parameters for each child view (including the child view group).

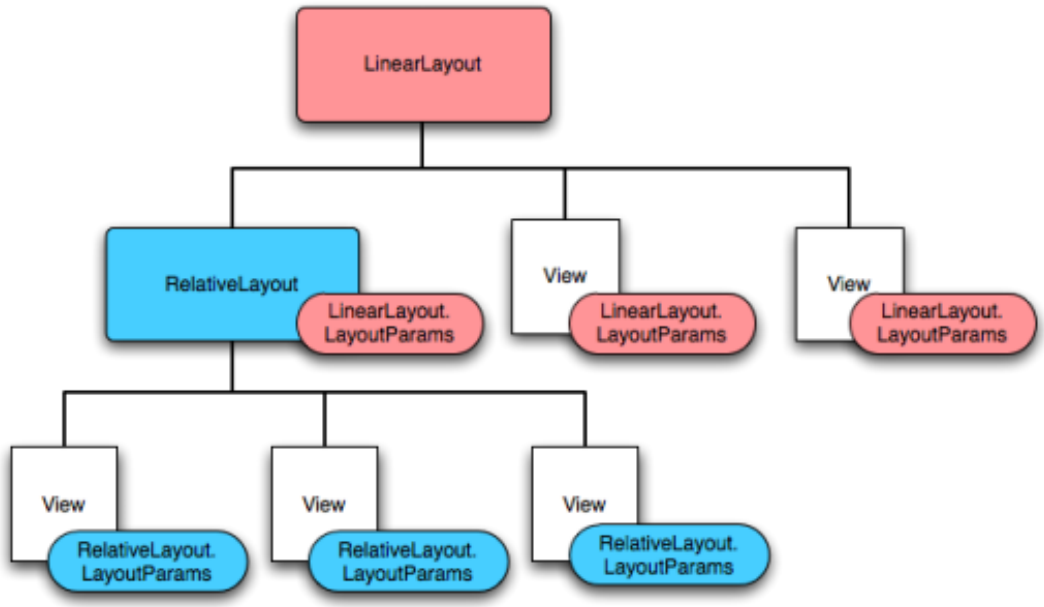

Figure 1. Visualization of a view hierarchy with layout parameters associated with each view.

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 use one of these constants to set the width or height:

- *wrap_content* tells your view to size itself to the dimensions required by its content
- *fill_parent* (renamed *match_parent* in API Level 8) tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (*dp*), *wrap_content*, or *fill_parent*, is a better approach, because it helps ensure that your application will display properly across a variety of device screen sizes. 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.

For more information about dimensions, see Dimension Values.

← Back to User Interface ↑ Go to top

Android 3.1 r1 - 17 Jun 2011 10:58