Building Custom Components

Android offers a sophisticated and powerful componentized model for building your UI, based on the fundamental layout classes: `View` and `ViewGroup`. To start with, the platform includes a variety of prebuilt View and ViewGroup subclasses — called widgets and layouts, respectively — that you can use to construct your UI.

A partial list of available widgets includes `Button`, `TextView`, `EditText`, `ListView`, `CheckBox`, `RadioButton`, `Gallery`, `Spinner`, and the more special-purpose `AutoCompleteTextView`, `ImageSwitcher`, and `TextSwitcher`.

Among the layouts available are `LinearLayout`, `FrameLayout`, `RelativeLayout`, and others. For more examples, see Common Layout Objects.

If none of the prebuilt widgets or layouts meets your needs, you can create your own View subclass. If you only need to make small adjustments to an existing widget or layout, you can simply subclass the widget or layout and override its methods.

Creating your own View subclasses gives you precise control over the appearance and function of a screen element. To give an idea of the control you get with custom views, here are some examples of what you could do with them:

- You could create a completely custom-rendered View type, for example a "volume control" knob rendered using 2D graphics, and which resembles an analog electronic control.
- You could combine a group of View components into a new single component, perhaps to make something like a ComboBox (a combination of popup list and free entry text field), a dual-pane selector control (a left and right pane with a list in each where you can re-assign which item is in which list), and so on.
- You could override the way that an EditText component is rendered on the screen (the Notepad Tutorial uses this to good effect, to create a lined-notepad page).
- You could capture other events like key presses and handle them in some custom way (such as for a game).

The sections below explain how to create custom Views and use them in your application. For detailed reference information, see the `View` class.

The Basic Approach

Here is a high level overview of what you need to know to get started in creating your own View components:

1. Extend an existing `View` class or subclass with your own class.
2. Override some of the methods from the superclass. The superclass methods to override start with '`on`', for example, `onDraw()`, `onMeasure()`, and `onKeyDown()`. This is similar to the `on...` events in `Activity` or `ListActivity` that you override for lifecycle and other functionality hooks.
3. Use your new extension class. Once completed, your new extension class can be used in place of the view upon which it was based.

> **Tip:** Extension classes can be defined as inner classes inside the activities that use them. This is useful because it controls access to them but isn't necessary (perhaps you want to create a new public View for wider use in your application).

Fully Customized Components

Fully customized components can be used to create graphical components that appear however you wish. Perhaps a graphical VU meter that looks like an old analog gauge, or a sing-a-long text view where a bouncing ball moves along the words so you can sing along with a karaoke machine. Either way, you want something that the built-in components just won't do, no matter how you combine them.

Fortunately, you can easily create components that look and behave in any way you like, limited perhaps only by your imagination, the size of the screen, and the available processing power (remember that ultimately your application might have to run on something with significantly less power than your desktop workstation).

To create a fully customized component:

1. The most generic view you can extend is, unsurprisingly, `View`, so you will usually start by extending this to create your new super component.
2. You can supply a constructor which can take attributes and parameters from the XML, and you can also consume your own such attributes and parameters (perhaps the color and range of the VU meter, or the width and damping of the needle, etc.)
3. You will probably want to create your own event listeners, property accessors and modifiers, and possibly more sophisticated behavior in your component class as well.
4. You will almost certainly want to override `onMeasure()` and are also likely to need to override `onDraw()` if you want the component to show something. While both have default behavior, the default `onDraw()` will do nothing, and the default `onMeasure()` will always set a size of 100x100 — which is probably not what you want.
5. Other `on...` methods may also be overridden as required.

Extend `onDraw()` and `onMeasure()`

The `onDraw()` method delivers you a `Canvas` upon which you can implement anything you want: 2D graphics, other standard or custom components, styled text, or anything else you can think of.

> **Note:** This does not apply to 3D graphics. If you want to use 3D graphics, you must extend `SurfaceView` instead of View, and draw from a separate thread. See the GLSurfaceViewActivity sample for details.

`onMeasure()` is a little more involved. `onMeasure()` is a critical piece of the rendering contract between your component and its container. `onMeasure()` should be overridden to efficiently and accurately report the measurements of its contained parts. This is made slightly more complex by the requirements of limits from the parent (which are passed in to the `onMeasure()` method) and by the requirement to call the `setMeasuredDimension()` method with the measured width and height once they have been calculated. If you fail to call this method from an overridden `onMeasure()` method, the result will be an exception at measurement time.

At a high level, implementing `onMeasure()` looks something like this:

1. The overridden `onMeasure()` method is called with width and height measure specifications (`widthMeasureSpec` and `heightMeasureSpec` parameters, both are integer codes representing dimensions) which should be treated as requirements for the restrictions on the width and height measurements you should produce. A full reference to the kind of restrictions these specifications can require can be found in the reference documentation under `View.onMeasure(int, int)` (this reference documentation does a pretty good job of explaining the whole measurement operation as well).
2. Your component's `onMeasure()` method should calculate a measurement width and height which will be required to render the component. It should try to stay within the specifications passed in, although it can choose to exceed them (in this case, the parent can choose what to do, including clipping, scrolling, throwing an exception, or asking the `onMeasure()` to try again, perhaps with different measurement specifications).
3. Once the width and height are calculated, the `setMeasuredDimension(int width, int height)` method must be called with the calculated measurements. Failure to do this will result in an exception being thrown.

Here's a summary of some of the other standard methods that the framework calls on views:

Category	Methods	Description
Creation	Constructors	There is a form of the constructor

		that are called when the view is created from code and a form that is called when the view is inflated from a layout file. The second form should parse and apply any attributes defined in the layout file.
	`onFinishInflate()`	Called after a view and all of its children has been inflated from XML.
Layout	`onMeasure(int, int)`	Called to determine the size requirements for this view and all of its children.
	`onLayout(boolean, int, int, int, int)`	Called when this view should assign a size and position to all of its children.
	`onSizeChanged(int, int, int, int)`	Called when the size of this view has changed.
Drawing	`onDraw(Canvas)`	Called when the view should render its content.
Event processing	`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.
Focus	`onFocusChanged(boolean, int, Rect)`	Called when the view gains or loses focus.
	`onWindowFocusChanged(boolean)`	Called when the window containing the view gains or loses focus.
Attaching	`onAttachedToWindow()`	Called when the view is attached to a window.
	`onDetachedFromWindow()`	Called when the view is detached from its window.
	`onWindowVisibilityChanged(int)`	Called when the visibility of the window containing the view has changed.

A Custom View Example

The CustomView sample in the API Demos provides an example of a customized View. The custom View is defined in the LabelView class.

The LabelView sample demonstrates a number of different aspects of custom components:

- Extending the View class for a completely custom component.
- Parameterized constructor that takes the view inflation parameters (parameters defined in the XML). Some of these are passed through to the View superclass, but more importantly, there are some custom attributes defined and used for LabelView.
- Standard public methods of the type you would expect to see for a label component, for example `setText()`, `setTextSize()`, `setTextColor()` and so on.

- An overridden `onMeasure` method to determine and set the rendering size of the component. (Note that in LabelView, the real work is done by a private `measureWidth()` method.)
- An overridden `onDraw()` method to draw the label onto the provided canvas.

You can see some sample usages of the LabelView custom View in custom_view_1.xml from the samples. In particular, you can see a mix of both `android:` namespace parameters and custom `app:` namespace parameters. These `app:` parameters are the custom ones that the LabelView recognizes and works with, and are defined in a styleable inner class inside of the samples R resources definition class.

Compound Controls

If you don't want to create a completely customized component, but instead are looking to put together a reusable component that consists of a group of existing controls, then creating a Compound Component (or Compound Control) might fit the bill. In a nutshell, this brings together a number of more atomic controls (or views) into a logical group of items that can be treated as a single thing. For example, a Combo Box can be thought of as a combination of a single line EditText field and an adjacent button with an attached PopupList. If you press the button and select something from the list, it populates the EditText field, but the user can also type something directly into the EditText if they prefer.

In Android, there are actually two other Views readily available to do this: `Spinner` and `AutoCompleteTextView`, but regardless, the concept of a Combo Box makes an easy-to-understand example.

To create a compound component:

1. The usual starting point is a Layout of some kind, so create a class that extends a Layout. Perhaps in the case of a Combo box we might use a LinearLayout with horizontal orientation. Remember that other layouts can be nested inside, so the compound component can be arbitrarily complex and structured. Note that just like with an Activity, you can use either the declarative (XML-based) approach to creating the contained components, or you can nest them programmatically from your code.
2. In the constructor for the new class, take whatever parameters the superclass expects, and pass them through to the superclass constructor first. Then you can set up the other views to use within your new component; this is where you would create the EditText field and the PopupList. Note that you also might introduce your own attributes and parameters into the XML that can be pulled out and used by your constructor.
3. You can also create listeners for events that your contained views might generate, for example, a listener method for the List Item Click Listener to update the contents of the EditText if a list selection is made.
4. You might also create your own properties with accessors and modifiers, for example, allow the EditText value to be set initially in the component and query for its contents when needed.
5. In the case of extending a Layout, you don't need to override the `onDraw()` and `onMeasure()` methods since the layout will have default behavior that will likely work just fine. However, you can still override them if you need to.
6. You might override other `on...` methods, like `onKeyDown()`, to perhaps choose certain default values from the popup list of a combo box when a certain key is pressed.

To summarize, the use of a Layout as the basis for a Custom Control has a number of advantages, including:

- You can specify the layout using the declarative XML files just like with an activity screen, or you can create views programmatically and nest them into the layout from your code.
- The `onDraw()` and `onMeasure()` methods (plus most of the other `on...` methods) will likely have suitable behavior so you don't have to override them.
- In the end, you can very quickly construct arbitrarily complex compound views and re-use them as if they were a single component.

Examples of Compound Controls

In the API Demos project that comes with the SDK, there are two List examples — Example 4 and Example 6 under Views/Lists demonstrate a SpeechView which extends LinearLayout to make a component for displaying Speech quotes. The corresponding classes in the sample code are `List4.java` and `List6.java`.

Modifying an Existing View Type

There is an even easier option for creating a custom View which is useful in certain circumstances. If there is a component that is already very similar to what you want, you can simply extend that component and just override the behavior that you want to change. You can do all of the things you would do with a fully customized component, but by starting with a more specialized class in the View hierarchy, you can also get a lot of behavior for free that probably does exactly what you want.

For example, the SDK includes a NotePad application in the samples. This demonstrates many aspects of using the Android platform, among them is extending an EditText View to make a lined notepad. This is not a perfect example, and the APIs for doing this might change from this early preview, but it does demonstrate the principles.

If you haven't done so already, import the NotePad sample into Eclipse (or just look at the source using the link provided). In particular look at the definition of `MyEditText` in the NoteEditor.java file.

Some points to note here

1. **The Definition**

   The class is defined with the following line:
   `public static class MyEditText extends EditText`

   - It is defined as an inner class within the `NoteEditor` activity, but it is public so that it could be accessed as `NoteEditor.MyEditText` from outside of the `NoteEditor` class if desired.
   - It is `static`, meaning it does not generate the so-called "synthetic methods" that allow it to access data from the parent class, which in turn means that it really behaves as a separate class rather than something strongly related to `NoteEditor`. This is a cleaner way to create inner classes if they do not need access to state from the outer class, keeps the generated class small, and allows it to be used easily from other classes.
   - It extends `EditText`, which is the View we have chosen to customize in this case. When we are finished, the new class will be able to substitute for a normal `EditText` view.
2. **Class Initialization**

   As always, the super is called first. Furthermore, this is not a default constructor, but a parameterized one. The EditText is created with these parameters when it is inflated from an XML layout file, thus, our constructor needs to both take them and pass them to the superclass constructor as well.
3. **Overridden Methods**

   In this example, there is only one method to be overridden: `onDraw()` — but there could easily be others needed when you create your own custom components.

   For the NotePad sample, overriding the `onDraw()` method allows us to paint the blue lines on the `EditText` view canvas (the canvas is passed into the overridden `onDraw()` method). The super.onDraw() method is called before the method ends. The superclass method should be invoked, but in this case, we do it at the end after we have painted the lines we want to include.
4. **Use the Custom Component**

   We now have our custom component, but how can we use it? In the NotePad example, the custom component is used directly from the declarative layout, so take a look at `note_editor.xml` in the `res/layout` folder.

   ```
   <view
     class="com.android.notepad.NoteEditor$MyEditText"
     id="@+id/note"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:background="@android:drawable/empty"
     android:padding="10dip"
     android:scrollbars="vertical"
     android:fadingEdge="vertical" />
   ```

   - The custom component is created as a generic view in the XML, and the class is specified using the full package. Note also that the inner class we defined is referenced using the `NoteEditor$MyEditText` notation which is a standard way to refer to inner classes in the Java programming language.

     If your custom View component is not defined as an inner class, then you can, alternatively, declare the View component with the XML element name, and exclude the `class` attribute. For example:

```
<com.android.notepad.MyEditText
  id="@+id/note"
  ... />
```

Notice that the `MyEditText` class is now a separate class file. When the class is nested in the `NoteEditor` class, this technique will not work.

- The other attributes and parameters in the definition are the ones passed into the custom component constructor, and then passed through to the EditText constructor, so they are the same parameters that you would use for an EditText view. Note that it is possible to add your own parameters as well, and we will touch on this again below.

And that's all there is to it. Admittedly this is a simple case, but that's the point — creating custom components is only as complicated as you need it to be.

A more sophisticated component may override even more `on...` methods and introduce some of its own helper methods, substantially customizing its properties and behavior. The only limit is your imagination and what you need the component to do.

← Back to User Interface ↑ Go to top