计算机科学与技术学院

毕业设计

英文翻译

论文题目	快速文件传输 APP 的设计与实现 ——手机与手机互传		
学校导师	余颖	职　称	讲师
企业导师	罗超	职　称	
企业名称	华信教育科技有限公司		
学生姓名	张涛	学　号	20134350215
专　业	软件工程	班　级	本 13 软件 02 班
系 主 任	刘杰	院　长	刘振宇
起止时间	2016 年 6 月 5 日至 2017 年 5 月 22 日		

2016 年 6 月 25 日

Custom Components

In this document

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:

Extend an existing View class or subclass with your own class.

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.

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:

The most generic view you can extend is, unsurprisingly, View, so you will usually start by extending this to create your new super component.
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.)
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.
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.
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:

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).

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).

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:

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.

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.

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.

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.

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.

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 Android Studio (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

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.

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.

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.

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.

自定义组件

Android 提供了一个复杂而强大的组件化模型，用于构建您的 UI，基于基本的布局类：View 和 ViewGroup。首先，平台包括各种与构建的 View 和 ViewGroup 子类，分别称为组件和布局，用于构建 UI（user interfere，用户界面）。

一部分的组件列表有 Button，TextView，Edittext，Listview，CheckBox，RadioBox，Gallery，Spinner 以及更专用的 AutoComplete TextView，ImageSwitcher 和 TextSwitcher。

可用的布局有 LinearLayout，FrameLayout，RelativeLayout 等。 有关更多示例，请参阅通用布局对象。

如果没有预先构建的小部件或布局满足您的需要，则可以创建自己的 View 子类。如果只需要对现有的窗口小部件或布局进行小的调整，那么可以简单地对窗口小部件或布局进行子类化，并覆盖其方法。

创建您自己的 View 子类可以精确控制屏幕元素的外观和功能。 通过自定义视图来了解您获得的控制，以下是您可以使用它们的一些示例：

1. 您可以创建一个完全自定义渲染的视图类型，例如使用 2D 图形渲染的“音量控制”旋钮，并且类似于模拟电子控件。
2. 您可以将一组视图组件组合到一个新的单个组件中，也许可以使一些类似 ComboBox（弹出列表和自由条目文本字段的组合），双窗格选择器控件（左窗格和右窗格，列表在 每个可以重新分配哪个项目在哪个列表中），等等。
3. 您可以覆盖 EditText 组件在屏幕上呈现的方式（记事本教程使用此功能获得良好效果，创建一个排序的记事本页面）。
4. 您可以捕获其他事件，例如按键的按压，并以某种自定义方式（如游戏）来处理它们。

以下部分介绍如何创建自定义视图并在应用程序中使用它们。 有关详细参考信息，请参阅 View 类。

基本方法

以下是创建自己的 View 组件所需的高度概述：

1. 使用自己的类扩展现有的 View 类或子类；
2. 覆盖超类中的一些方法。超类的方法以“on”开头，例如 onDraw()，onMeasure

（）和 onKeyDown（））。 这与 Activity 或 ListActivity 中的 on...事件类似，您将为生命周期和其他功能钩子覆盖。

3. 使用你的新扩展类。一旦完成，您的新扩展类可以用于代替基于它的视图。

提示：扩展类可以在使用它们的活动中定义为内部类。 这是有用的，因为它控制对它们的访问，但不是必需的（也许您想要创建一个新的公共视图，以便在您的应用程序中更广泛地使用）

完全定制的组件

完全自定义的组件可用于创建出现的图形组件。 也许是一个图形 VU 表，看起来像一个旧的模拟量表，或一个长长的文本视图，弹跳球沿着这些词移动，所以你可以和一个卡拉 OK 机一起唱歌。 无论哪种方式，你想要的东西，内置的组件不会做，无论你如何组合它们。

幸运的是，您可以轻松地以任何您喜欢的方式创建看起来和行为的组件，也许只有您的想象力，屏幕尺寸和可用的处理能力（请记住，最终您的应用程序可能必须运行在显着较少的东西上） 电源比您的桌面工作站）。

要创建一个完全自定义的组件：

1. 您可以扩展的最通用的视图是，令人惊讶的是，View，所以通常开始扩展它来创建新的超级组件。
2. 您可以提供一个可以从 XML 中获取属性和参数的构造函数，还可以使用自己的这些属性和参数（也可以使用 VU 表的颜色和范围，或针的宽度和阻尼等）
3. 您可能希望在组件类中创建自己的事件侦听器，属性访问器和修饰符，以及可能更复杂的行为。
4. 您几乎肯定要覆盖 onMeasure（），如果您希望组件显示某些东西，也可能需要重写 onDraw（）。 虽然两者都有默认行为，但是默认的 onDraw（）将不会执行任何操作，默认的 onMeasure（）将始终设置为 100x100 的大小 - 这可能不是您想要的。
5. 其他的...方法也可以根据需要被覆盖。

扩展 onDraw（）和 onMeasure（）

onDraw（）方法提供了一个 Canvas，您可以在其上实现所需的任何内容：2D 图形，其他标准或自定义组件，样式文本或任何您可以想到的内容。

注意：这不适用于 3D 图形。 如果要使用 3D 图形，则必须扩展 SurfaceView 而不是 View，并从单独的线程绘制。 有关详细信息，请参阅 GLSurfaceViewActivity 示例。

onMeasure（）有一点更多的参与。 onMeasure（）是您的组件与其容器之间的渲染契约的关键部分。 onMeasure（）应该被覆盖，以有效和准确地报告其包含的部分的测量。 通过父级（传递给 onMeasure（）方法）的限制要求，以及一旦计算

出测量的宽度和高度，就调用 setMeasuredDimension（）方法的要求，这一点稍微复杂一些。 如果您无法从覆盖的 onMeasure（）方法调用此方法，则结果将在测量时间发生异常。

在高水平上，实现 onMeasure（）看起来像这样：

1. 被调用的 onMeasure（）方法被调用宽度和高度度量规格（widthMeasureSpec 和 heightMeasureSpec 参数，都是代表尺寸的整数代码），它们应该被视为对应该产生的宽度和高度测量的限制的要求。 参考这些规范可以要求的限制类型，可以在 View.onMeasure（int，int）的参考文档中找到（本参考文档在解释整个测量操作方面做得很好）。
2. 您的组件的 onMeasure（）方法应该计算渲染组件所需的测量宽度和高度。 它应该尽量保持在传递的规范内，尽管它可以选择超出它们（在这种情况下，父级可以选择做什么，包括剪切，滚动，抛出异常，或者要求 onMeasure（）再次尝试，也许具有不同的测量规格）。
3. 一旦计算了宽度和高度，就必须使用计算的度量来调用 setMeasuredDimension（int width，int height）方法。 否则将导致抛出异常。

以下是框架调用视图的其他一些标准方法的总结：

分类	方法	描述
创建	构造函数	有一种构造函数的形式，当视图是从代码创建的，并且当视图从布局文件中膨胀时调用该窗体。第二种形式应解析并应用布局文件中定义的任何属性。
	onFinishInflate（）	在视图之后被调用，并且所有的孩子都已

		经从xml 中 映 射 了
布局	onMeasure(int, int)	用于确定此视图及其所有子项的大小要求。
	onLayout(boolean, int, int, int, int)	当此视图应该为其所有子项分配大小和位置时调用。
	onSizeChanged（int，int，int，int）	当此视图的大小更改时调用。
绘制	onDraw(Canvas)	当视图呈现其内容时调用。
事件处理	onKeyDown（int，KeyEvent）	当发生新的键事件时调用
	onKeyUp（int，KeyEvent）	当启动事件发生时调用
	onTrackballEvent（MotionEvent）	发生轨迹球运动事件时调用。
	onTouchEvent（MotionEvent）	触发屏幕运动事件发生时调用
焦点	onFocusChanged(boolean,int,Rect）	当视图增益或失去焦点时调用
	onWindowFocusChanged（boolean）	当包含视图的窗口增益或失去焦点时调用
附加	onAttachedToWindow（）	当视图附加到窗口时调用
	onDetachedFromWindow()	当视图从其窗口分

		离时调用。
	onWindowVisibilityChanged（int）	当包含视图的窗口的可视性已更改时调用

一个例子

API Demos 中的CustomView示例提供了自定义视图的示例。CustomView在LabelView类中定义。

LabelView 示例演示了自定义组件的许多不同方面：

1. 扩展一个完全自定义组件的 View 类。
2. 参数化构造函数，用于查看通货膨胀参数（XML 中定义的参数）。 其中一些被传递给 View 超类，但更重要的是，有一些定义和用于 LabelView 的自定义属性。
3. 标签公共方法的类型，你会期望看到一个标签组件，例如 setText(),setTextSize（）, setTextColor（）等等。
4. 一种重写的onMeasure方法来确定和设置组件的渲染大小。(请注意，在LabelView中，实际工作是通过私有的 measureWidth（）方法完成的。)
5. 重写的 onDraw（）方法将标签绘制到所提供的画布上。

您可以在示例的 custom_view_1.xml 中看到 LabelView 自定义视图的一些示例用法。特别是，您可以看到 Android：命名空间参数和自定义应用程序：命名空间参数的混合。 这些 app：参数是 LabelView 识别和使用的自定义参数，并且在样本 R 资源定义类中的一个风格的内部类中定义。

符合控制

如果您不想创建一个完全自定义的组件，而是希望将由一组现有控件组成的可重用组件放在一起，则创建一个复合组件（或复合控件）可能适合该帐单。 简而言之，这将一些更多的原子控制（或视图）汇集成一系列可以被视为单一事物的逻辑组合。 例如，组合框可以被认为是单行 EditText 字段和具有附加 PopupList 的相邻按钮的组合。 如果您按下按钮并从列表中选择某些内容，则会填充 EditText 字段，但如果用户喜欢，用户还可以直接在 EditText 中键入内容。

在 Android 中，实际上还有两个其他视图可以实现：Spinner 和 AutoCompleteTextView，但无论 Combo Box 的概念是一个容易理解的例子。

创建复合组件：

1. 通常的起点是某种布局，因此创建一个扩展布局的类。 也许在组合框的情况下，我们可能会使用一个水平方向的 LinearLayout。 请记住，其他布局可以嵌套在内部，因此复合组件可以是任意复杂和结构化的。 请注意，就像使用 Activity 一样，您可以使用声明式（基于 XML）的方法来创建包含的组件，也可以使用编程方式将其嵌套在代码中。
2. 在新类的构造函数中，使用超类预期的任何参数，并将其传递给超类构造函数。 然后，您可以设置其他视图以在新组件中使用; 这是您创建 EditText 字段和 PopupList 的位置。 请注意，您还可以将自己的属性和参数引入可以由构造函数提取和使用的 XML。
3. 您还可以为包含的视图可能生成的事件创建侦听器，例如，如果创建了列表选择，则列表项单击侦听器的侦听器方法可更新 EditText 的内容。
4. 您还可以使用访问器和修饰符创建自己的属性，例如，允许最初在组件中设置 EditText 值，并在需要时查询其内容。
5. 在扩展 Layout 的情况下，您不需要重写 onDraw（）和 onMeasure（）方法，因为布局将具有可能正常工作的默认行为。 但是，如果需要，您仍然可以覆盖它们。
6. 您可以覆盖其他 on...方法，如 onKeyDown（），以便在按下某个键时从组合框的弹出列表中选择某些默认值。

总而言之，使用“布局”作为自定义控件的基础具有许多优点，包括：

-您可以使用声明性 XML 文件指定布局，就像使用活动屏幕一样，或者您可以以编程方式创建视图，并将其嵌入到代码中的布局中。

-onDraw（）和 onMeasure（）方法（加上其他大部分的...方法）可能会有合适的行为，所以你不必重写它们。

-最后，您可以很快地构建任意复杂的复合视图，并重新使用它们，就像它们是单个组件一样。

复合控制的例子

在 SDK 附带的 API Demos 项目中，有两个列表示例 - 视图/列表下的示例 4 和示例 6 演示了一个 SpeechView，它扩展了 LinearLayout 以创建显示语音引号的组件。

示例代码中的相应类是 List4.java 和 List6.java

修改现有视图类型

有一个更容易的选择是创建一个在某些情况下有用的自定义视图。如果有一个已经非常类似于你想要的组件，你可以简单地扩展该组件，并且只是覆盖你想要改变的行为。您可以使用完全自定义的组件执行所有操作，但是从 View 层次结构中的更专门的类开始，您还可以免费获取大量可能完全符合您想要的行为。

例如，SDK 中的样本中包含一个 NotePad 应用程序。这展示了使用 Android 平台的许多方面，其中包括扩展一个 EditText 视图，使一个内衬的记事本。这不是一个完美的例子，这样做的 API 可能会从这个早期的预览中改变，但它确实说明了原理。

如果您还没有这样做，请将 NotePad 样本导入 Android Studio（或使用提供的链接查看源代码）。特别看看 NoteEditor.java 文件中 MyEditText 的定义。

这里有一些要点：

1. 定义

该类定义如下：

public static class MyEditText 扩展了 EditText

1) 它被定义为 NoteEditor 活动中的内部类，但它是公共的，因此如果需要，可以从 NoteEditor 类的外部作为 NoteEditor.MyEditText 访问。
2) 它是静态的，这意味着它不会产生所谓的“合成方法”，允许它从父类访问数据，这反过来意味着它真的表现为一个单独的类，而不是与 NoteEditor 强烈相关的东西。 如果不需要从外部类访问状态，使生成的类保持较小，并允许其从其他类轻松使用，则这是创建内部类的更为清晰的方法。
3) 它扩展了 EditText，这是我们在这种情况下选择定制的视图。 完成后，新课程将可以替换普通的 EditText 视图。

2. 类的初始化

一如以往，超类被首先调用。此外，这不是一个默认构造函数，而是一个参数化的构造函数。 当使用这些参数从 XML 布局文件中充实时，使用这些参数创建 EditText，因此，我们的构造函数需要将它们传递给超类构造函数。

3. 重写方法

在此示例中，只有一种方法可以被覆盖：onDraw（） - 但是当您创建自己的自定义组件时，可能很容易需要其他方法。

对于 NotePad 示例，覆盖 onDraw（）方法允许我们在 EditText 视图画布上绘制蓝线（画布被传递到覆盖的 onDraw（）方法）。 在方法结束之前调用 super.onDraw（）方法。 应该调用超类方法，但在这种情况下，我们在绘制我们要包括的行之后最后再做。

4. 使用自定义组件

我们现在有了我们的定制组件，但是我们如何使用它？ 在 NotePad 示例中，自定义组件直接从声明性布局使用，因此请查看 res / layout 文件夹中的 note_editor.xml。

```
<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" />
```

a) 自定义组件被创建为 XML 中的通用视图，并且使用完整的包指定类。 还要注意，我们定义的内部类是使用 NoteEditor $ MyEditText 符号来引用的，这是一种标准方法来引用 Java 编程语言中的内部类。

如果您的自定义 View 组件未定义为内部类，那么您也可以使用 XML 元素名声明 View 组件，并排除该类属性。 例如：

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

请注意，MyEditText 类现在是一个单独的类文件。 当类嵌套在 NoteEditor 类中时，此技术将无法正常工作。

b) 定义中的其他属性和参数是传递给自定义组件构造函数的属性和参数，然后传递给 EditText 构造函数，因此它们与 EditText 视图使用的参数相同。 请

注意，您也可以添加自己的参数，下面将再次介绍。

这就是它的一切。 诚然，这是一个简单的例子，但这就是要点 - 创建自定义组件只是你需要的那么复杂。

更复杂的组件可以覆盖更多的...方法，并介绍一些自己的帮助方法，大大定制其属性和行为。 唯一的限制是你的想象力和你需要组件做什么。